MICROSOFT®

The High Performance Software™

About the Microsoft C Compiler

Welcome to the Microsoft® C Compiler for MS™-DOS. Microsoft C is a full implementation of the C language, a language known for its efficiency, economy, and portability. The Microsoft C Compiler provides a simple command structure with a flexible set of options to accommodate all levels of programming experience.

With the Microsoft C Compiler, you can take advantage of C's strengths. Three different memory models are defined to let you set up your program in the most efficient way, taking advantage of the segmented architecture of the Intel® 8086 family of processors. In addition, the Microsoft C Compiler lets you combine features from different memory models in "mixed model" programs. Mixed model programs let you change addressing conventions for one or more program items without changing the addressing conventions for the program as a whole.

Included with your C compiler is a set of more than 200 run-time library routines that provides you with an extensive base of built-in functions for use in your C programs. The MS-DOS C run-time library is designed to make writing portable programs easier by providing compatibility with the XENIX® run-time library for 80286 systems.

In fact, compatibility with the 286 XENIX operating system is a built-in feature of the Microsoft C Compiler for MS-DOS. This compiler shares its design with the 286 XENIX C compiler, written by Microsoft Corporation and chosen by IBM® for its Personal Computer AT.

The Microsoft C Compiler offers advanced optimizing capabilities. Optimizing is performed automatically whenever you compile. Command line options are available to select alternative optimizing procedures or to turn off optimizing in the early stages of program development.

The compiler generates a broad range of error and warning messages to help you locate errors and potential problems. A special command line option lets you adjust the level of warning messages to suit your own needs.

Package Contents

Your Microsoft C Compiler package contains the following programs, stored on floppy disks.

- The compiler software
- The Microsoft LINK utility
- The Microsoft LIB utility
- EXEPACK, the executable file compression utility
- EXEMOD, the executable file header modification utility

Two documentation binders are included with the package.

System Requirements

To use the Microsoft C Compiler, your machine must run MS-DOS Version 2.0 or later. You must have two double-sided disk drives and a minimum of 256K (kilobytes) of memory (one kilobyte is 1,024 bytes). You must use Microsoft LINK Version 3.0 or later (included in this package). You cannot use earlier versions of Microsoft LINK with the compiler.

About These Manuals

The two documentation binders in your Microsoft C Compiler package hold the three manuals listed below.

Microsoft C Compiler User's Guide

The *C User's Guide* gives you the information you need to set up and operate the Microsoft C Compiler on your computer and explains how to compile, link, and run your C programs. Refer to the *C User's Guide* when you have questions about invoking the compiler and linker or about this particular implementation of C on MS-DOS.

Microsoft C Language Reference

The *C Language Reference* defines the C language as implemented by Microsoft. Use the *C Language Reference* when you have questions about the rules and behavior of the C language.

Microsoft C Run-Time Library Reference

The *C Library Reference* describes the run-time library routines provided for use in your C programs. The first part of the *C Library Reference* gives an overview of the run-time library, while the second section presents the routines in alphabetical order for quick reference.

User's Guide

User's Guide

Microsoft®
C Compiler

User's Guide

for the MS™-DOS Operating System

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985

If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Microsoft, the Microsoft logo, and XENIX are registered trademarks, and MS is a trademark of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a trademark of Bell Laboratories.

Document Number 8415L-300-02
Part Number 048-014-025

Contents

1 Introduction 1

1.1 Overview 3
1.2 About This Manual 4
1.3 Notational Conventions 6
1.4 Learning More About C 9

2 Getting Started 11

2.1 Introduction 13
2.2 Backing Up Your Disks 13
2.3 Disk Contents 13
2.4 Understanding the Compiler Software 16
2.5 Setting Up the Environment 20
2.6 Setting Up Your CONFIG.SYS File 23
2.7 Using an 8087 or 80287 Coprocessor 24
2.8 Using an 80186, 80188, or 80286 Processor 24
2.9 Converting Existing C Programs 24
2.10 Organizing Your Software 25
2.11 Practice Session 31
2.12 Using Batch Files 36

3 Compiling 39

3.1 Introduction 41
3.2 Running the Compiler 42
3.3 Naming the Object File 50
3.4 Producing Listing Files 52
3.5 Controlling the Preprocessor 54
3.6 Syntax Checking 60
3.7 Selecting Floating-Point Options 63
3.8 Using 80186, 80188, or 80286 Processors 68
3.9 Understanding Error Messages 68
3.10 Preparing for Debugging 72
3.11 Optimizing 73
3.12 Compiling Large Programs 75
3.13 Working with Memory Models 76

4 Linking 81

4.1 Introduction 83
4.2 How the Linker Works 83
4.3 Linking C Program Files 84
4.4 Running the Linker 87
4.5 Controlling the Linker 94

5 Running Your C Program 109

5.1 Running a Program 111
5.2 Passing Data to a Program 111
5.3 Expanding Wild Card Arguments 114
5.4 Suppressing Command Line Processing 115
5.5 Suppressing Null Pointer Checks 116

6 Managing Libraries 119

6.1 Introduction 121
6.2 Overview of LIB Operation 122
6.3 Running LIB 123
6.4 Library Tasks 130

7 Advanced Topics 135

7.1 Introduction 137
7.2 Enabling Special Keywords 137
7.3 Packing Structure Members 137
7.4 Restricting Length of External Names 138
7.5 Labeling the Object File 139
7.6 Suppressing Default Library Selection 139
7.7 Controlling Floating-Point Operations 140
7.8 Advanced Optimizing 143
7.9 Modifying the Executable File 145
7.10 Controlling Binary and Text Modes 147
7.11 Mixed Model Programming 148
7.12 Setting the Data Threshold 154
7.13 Naming Modules and Segments 155

8 Interfaces with Other Languages 157

8.1 Assembly Language Interface 159
8.2 Calling FORTRAN and Pascal Routines 170

Appendices 173

A ASCII Character Codes 175

B Command Summary 177

B.1 Introduction 179
B.2 Compiler Summary 179
B.3 Linker Summary 184
B.4 The LIB Utility 187
B.5 The EXEPACK Utility 187
B.6 The EXEMOD Utility 188

C The CL Command 189

C.1 Introduction 191
C.2 Command Syntax and Options 191
C.3 Linking with the CL Command 195
C.4 Additional Options 196
C.5 XENIX-Compatible Options 196

D Converting from Previous Versions of the Compiler 199

D.1 Introduction 201
D.2 Language Definition Differences 201
D.3 Run-Time Library Differences 206
D.4 Differences in Assembly Language Interface 212

E Error Messages 223

E.1 Introduction 225
E.2 Run-Time Error Messages 225
E.3 Compiler Error Messages 230
E.4 Linker Error Messages 255
E.5 Library Manager Error Messages 261
E.6 EXEPACK Error Messages 263
E.7 EXEMOD Error Messages 264

F Working with Microsoft Products 265

F.1 Introduction 267
F.2 Microsoft LINK, Microsoft LIB, EXEPACK, and EXEMOD 267
F.3 286 XENIX Operating System 268
F.4 Microsoft FORTRAN and Microsoft Pascal 268
F.5 Microsoft Macro Assembler (MASM) and Symbolic Debug Utility (SYMDEB) 268

G Microsoft LINK Technical Summary 273

G.1 Introduction 275
G.2 Alignment of Segments 275
G.3 Frame Addresses 276
G.4 Order of Segments 276
G.5 Combined Segments 277
G.6 Groups 278
G.7 Fix-ups 279
G.8 Controlling the Loading Order 280

Index 283

Figures

Figure 8.1 Segment Setup in C Programs 160
Figure D.1 Version 2.03 Stack Frame Setup 215
Figure D.2 Version 3.0 Stack Frame Setup 216
Figure D.3 Version 2.03 S and P Model Layout 220
Figure D.4 Version 3.0 Layout 221

Tables

Table 3.1 Warning Levels 71

Table 7.1 Uses of near and far Keywords with Small Model 150

Table 7.2 Segment Naming Conventions 156

Table 8.1 Segments, Groups, and Classes for Standard Memory Models 164

Table 8.2 C Return Value Conventions 167

Table B.1 Text and Data Segments in Standard Memory Models 183

Table B.2 Pointer and Integer Sizes in Standard Memory Models 183

Table B.3 Segment Names in Standard Memory Models 184

Table C.1 Summary of /F Options 193

Table C.2 Arguments to /F Options 194

Table C.3 XENIX Options Accepted by the CL Command 197

Table E.1 Program Limits at Run Time 229

Table E.2 Limits Imposed by the C Compiler 254

Chapter 1
Introduction

1.1 Overview 3
1.2 About This Manual 4
1.3 Notational Conventions 6
1.4 Learning More About C 9

1.1 Overview

The C language is a powerful general-purpose programming language that is capable of generating efficient, compact and portable code. The Microsoft® C Compiler for the MS™-DOS operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*. Microsoft Corporation is actively involved in the development of the ANSI (American National Standards Institute) standard for the C language; this version of Microsoft C attempts to anticipate and conform to the forthcoming standard.

Microsoft C offers several important features to help you increase the efficiency of your C programs. You can choose between three standard memory models (small, medium and large) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the Microsoft C Compiler allows you to "mix" memory models by using special declarations in your program.

The C language does not provide such standard features as input and output capabilities and string manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the C installation. Because the functions that require interaction with the operating system (for example, input and output) are logically separate from the language itself, the C language is especially suited for producing portable code.

The portability of your Microsoft C programs is increased by the use of a common run-time library for MS-DOS and XENIX® installations. Using the routines in this library you can easily transport programs from a XENIX development environment to an MS-DOS machine, or vice versa. See the *Microsoft C Run-Time Library Reference* (included in this package) for more information on the common library for MS-DOS and XENIX.

Compared to other programming languages, C is extremely flexible about data conversions and nonstandard constructions. The Microsoft C Compiler offers several levels of warnings to help you control this flexibility. Programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. The experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions.

1.2 About This Manual

This manual explains how to use the Microsoft C Compiler to compile, link, and run C programs on your MS-DOS system. The manual assumes that you are familiar with the C language and with MS-DOS and that you are able to create and edit a C language source file on your system. If you have questions about the C language, turn to the *Microsoft C Language Reference*, included in this package. The *Microsoft C Run-Time Library Reference* documents the run-time library routines you can use in your C programs. For more information about C, refer to Section 1.4, "Learning More About C." A brief description of the remaining chapters of the *C User's Guide* is given below.

Chapter 2, "Getting Started," covers installing and organizing the compiler software. This chapter explains how to set up an operating environment for the compiler by defining environment variables and includes a practice session to acquaint you with the Microsoft C Compiler.

Chapter 3, "Compiling," discusses the process of compiling a program using the basic compiler command MSC. This chapter contains a detailed description of the options most commonly used to control preprocessing, compilation, and output of files. The standard memory models (small, medium, and large) are discussed in this chapter.

Chapter 4, "Linking," describes the Microsoft LINK Object Code Linker and the options available to control its operation. This chapter includes a discussion of the special requirements that apply when linking C program files.

Chapter 5, "Running Your C Program," explains how to run your executable program file and how to pass data to a program at execution time.

Chapter 6, "Managing Libraries," describes the Microsoft LIB Library Manager. This utility enables you to create and maintain your own libraries of useful functions. You can use these libraries to customize the run-time support available to your programs.

Chapter 7, "Advanced Topics," describes additional command line options for the experienced programmer and gives the technical information necessary to use them. "Mixed model" programming (combining features from the three standard memory models) is discussed in this chapter, as well as the EXEPACK and EXEMOD utilities, which can be used to modify executable files.

Chapter 8, "Interfaces with Other Languages," covers two main topics: the interface between assembly language routines and C routines, and declarations of Microsoft Pascal and FORTRAN routines in C programs.

The appendices at the end of this manual contain useful reference material. Appendix A, "ASCII Character Codes," gives the ASCII decimal, octal, and hexadecimal equivalents for characters.

Appendix B, "Command Summary," provides a complete list of command line options for the MSC command and summarizes characteristics of the small, medium and large memory models. It also summarizes command characters and options for Microsoft LINK, Microsoft LIB, and the EXEPACK and EXEMOD utilities.

Appendix C, "The CL Command," describes an alternate command for invoking the compiler, the CL command. This command provides an interface that is similar to the XENIX and UNIX™ "cc" command.

Appendix D, "Converting from Previous Versions of the Compiler," summarizes the differences between this version of the Microsoft C Compiler and previous versions. This appendix gives instructions for converting your existing programs to work under Version 3.0 and later.

Appendix E, "Error Messages," lists and describes the error messages generated by the compiler, the linker, and the library manager. It also lists and explains run-time error messages.

Appendix F, "Working with Microsoft Products," gives an overview of Microsoft products included in the C compiler package and in other Microsoft language packages, and explains how these products work together. In particular, it demonstrates how to use SYMDEB, the Microsoft Symbolic Debug Utility (not included in this package) with C programs.

Appendix G, "Microsoft LINK Technical Summary," is a technical discussion of the linker's operation.

1.3 Notational Conventions

The following notational conventions are used throughout this manual.

italics

Italics mark the places in command line and option specifications and in the text where specific terms appear in an actual command. For example, in

/W number

number is italicized to indicate that this is a general form for the /W option. In an actual command, the user supplies a particular number for the placeholder *number*.

Italics are also used when referring to specific identifiers supplied for functions, variables, types, and labels. For instance, when a program example such as

```
pc = &count;
```

is provided, the variable names *pc* and *count* are italicized in the discussion of the example.

Occasionally, italics are used to emphasize particular words in the text.

[brackets]

Brackets enclose optional fields in command line and option specifications. For example, in

/D identifier[=[*string*]]

the brackets around the phrase "= [*string*]" indicate that you are not required to supply this phrase when you use the /D option. Furthermore, within this phrase, *string* is enclosed in brackets. Thus, when you give an equal sign (=), the *string* is optional. Notice, however, that you may not give a *string* without first giving the equal sign.

The C language also uses brackets for array declarations and subscript expressions. In

examples, brackets have the meaning specified by C. For instance,

```
a[10]
```

is an example showing a C subscript expression.

ellipses...

Ellipses following an item indicate that more items having the same form may appear. For example, in

LIB *library* [*/pagesize*] *operations*...

the ellipses indicate that one or more *operations* are allowed.

Vertical ellipses are also used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, three statements are shown. The ellipses between the statement indicate that intervening program lines occur but are not shown.

```
count = 0;
   .
   .
   .
*pc++;
   .
   .
   .
count = 0;
```

CAPITALS

Capital letters are used for the names of files, directories, environment variables, and manifest constants and macros. Commands typed at the MS-DOS level are also capitalized. These commands include built-in MS-DOS commands such as SET, as well as the MSC, LINK, and LIB commands, which invoke the compiler, linker, and library manager programs. You are not required to use capital letters when you actually enter these commands.

For example, in the command

```
SET TMP=B:\SCRATCH
```

the MS-DOS command SET is capitalized, as is the environment variable name TMP and the directory name to which it is set, B:\SCRATCH.

SMALL CAPITALS — Small capital letters are used for the names of keys and key sequences, such as RETURN and CONTROL-C.

"quotation marks" — Quotation marks set off terms defined in the text. For example, the term "far" appears in quotation marks the first time it is defined.

Quotation marks are also used to set off program fragments and to refer to command line prompts. For example, Microsoft LINK prompts you for the name of the executable file; this prompt is referred to as the "Run File" prompt.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than " ". For example,

```
"abc"
```

is a C string.

keywords — C keywords, such as **goto** and **char**, are set in a different type font (the Helvetica font) to distinguish them from ordinary identifiers and text.

programming examples — Programming examples are displayed without proportional spacing so that they look similar to the programs you create with a text editor.

1.4 Learning More About C

The three manuals in this documentation package provide a complete programmer's reference for Microsoft C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one of the following books.

Hancock, Les and Morris Krieger. *The C Primer.* New York: McGraw-Hill Book Co., Inc., 1982.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C.* Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas. *Learning to Program in C.* Cardiff, New Jersey: Plum Hall, Inc., 1983.

Chapter 2
Getting Started

2.1 Introduction 13
2.2 Backing Up Your Disks 13
2.3 Disk Contents 13
2.4 Understanding the Compiler Software 16
2.4.1 Executable Files 16
2.4.2 Include Files 17
2.4.3 Library Files 17
2.4.4 Other Files 19
2.5 Setting Up the Environment 20
2.6 Setting Up Your CONFIG.SYS File 23
2.7 Using an 8087 or 80287 Coprocessor 24
2.8 Using an 80186, 80188, or 80286 Processor 24
2.9 Converting Existing C Programs 24
2.10 Organizing Your Software 25
2.10.1 Sample Hard Disk Setup 26
2.10.2 Sample Floppy Disk Setup 27
2.11 Practice Session 31
2.12 Using Batch Files 36

2.1 Introduction

This chapter explains how to install the compiler software and set up an operating environment for the compiler. It describes the files that make up your compiler package and suggests methods for organizing the files.

Several MS-DOS procedures are mentioned in this chapter. In particular, the MS-DOS SET and PATH commands are used to give values to "environment variables," which control the compiler environment. If you are unfamiliar with the SET and PATH commands or with other MS-DOS procedures mentioned in this chapter, consult your MS-DOS manual for instructions.

This chapter includes a sample disk setup for your files and a practice session to introduce you to the process of compiling and linking a program with the Microsoft C Compiler and Microsoft LINK utility. The practice session, while not required, allows you to confirm that your files are set up properly and provides a quick overview of the MSC and LINK commands.

2.2 Backing Up Your Disks

The first thing you should do when you have unwrapped your system disks is to make working copies, using the COPY or DISKCOPY utility supplied with MS-DOS. Save the original disks for backup.

2.3 Disk Contents

When you first open your compiler package, you may want to verify that you have a complete set of software. You should find the following files on your disks.

Executable Files

Filename	Description
MSC.EXE	Control program for the compiler
P0.EXE	Preprocessor
P1.EXE	Language parser
P2.EXE	Code generator
P3.EXE	Optimizer, link text emitter, and assembly listing generator
LINK.EXE	Linker utility, Microsoft LINK
LIB.EXE	Library manager utility, Microsoft LIB
EXEPACK.EXE	File compression utility
EXEMOD.EXE	File header modification utility
CL.EXE	Alternate control program for the compiler

Include Files

Filename	Description
ASSERT.H	Defines *assert* macro
CONIO.H	Declares console I/O functions
CTYPE.H	Defines character classification macros
DIRECT.H	Declares directory control functions
DOS.H	Defines data types and macros for DOS interface functions and declares DOS interface functions
ERRNO.H	Defines system-wide error numbers
FCNTL.H	Defines flags used in *open* functions
IO.H	Declares functions that work on file handles ("low-level" functions)
MALLOC.H	Declares memory allocation functions
MATH.H	Declares math functions and defines related constants
MEMORY.H	Declares buffer manipulation functions
PROCESS.H	Declares process control functions and defines flags for *spawn* functions
SEARCH.H	Declares searching and sorting functions
SETJMP.H	Declares and sets up storage for *setjmp* and *longjmp* functions
SHARE.H	Defines flags for file sharing

SIGNAL.H	Declares *signal* function and defines related constants
STDIO.H	Declares stream functions and defines related macros, constants, and types
STDLIB.H	Declares all functions from the C run-time library that are not declared in other include files
STRING.H	Declares string manipulation functions
TIME.H	Declares time functions and defines structure type used by time functions
V2TOV3.H	Defines macros to aid in converting programs from Microsoft C Versions 2.03 and earlier
SYS\LOCKING.H	Defines flags for file locking
SYS\STAT.H	Declares *stat* and *fstat* functions and defines *stat* structure type and related constants
SYS\TIMEB.H	Declares *ftime* function and defines the *timeb* structure type
SYS\TYPES.H	Defines types used for file status and time information
SYS\UTIME.H	Declares *utime* function and defines the *utimbuf* structure type

Library Files

Filename	Description
SLIBC.LIB	Small model standard C library
SLIBFP.LIB	Small model floating-point math library
SLIBFA.LIB	Small model alternate math library
MLIBC.LIB	Medium model standard C library
MLIBFP.LIB	Medium model floating-point math library
MLIBFA.LIB	Medium model alternate math library
LLIBC.LIB	Large model standard C library
LLIBFP.LIB	Large model floating-point math library
LLIBFA.LIB	Large model alternate math library
EM.LIB	Emulator floating-point library
87.LIB	8087/80287 floating-point library

Other Files

Filename	Description
BINMODE.OBJ	Routine for processing binary data.
SSETARGV.OBJ	Small model routine for processing wild card characters.
MSETARGV.OBJ	Medium model routine for processing wild card characters.
LSETARGV.OBJ	Large model routine for processing wild card characters.
DEMO.C	Sample C program.
README.DOC	Documentation of most recent changes and additions not appearing in this manual. If you see files on your disks that do not appear in the above list, they will be explained in the README.DOC file. Your release of the software may not include a README.DOC file, so don't be alarmed if you are unable to find this file on your disks.

2.4 Understanding the Compiler Software

The software for the Microsoft C Compiler consists of three main categories of files: compiler executable files, include files, and library files. These files are listed in Section 2.3, "Disk Contents." Sections 2.4.1, 2.4.2, and 2.4.3, respectively, describe each of the three file categories in more detail. A number of additional files do not fall into the three main categories and are discussed separately in Section 2.4.4, "Other Files."

2.4.1 Executable Files

Executable files have an ".EXE" extension. MSC.EXE, the control program for the compiler, is an executable file. To run the compiler, invoke MSC.EXE by typing "MSC" or "msc".

P0.EXE, P1.EXE, P2.EXE, and P3.EXE are the four stages, or "passes," of the compiler. They are executed in order when you process a file using the compiler control program (MSC.EXE or CL.EXE).

The file LINK.EXE is the linker utility, Microsoft LINK. You invoke the linker by typing "LINK", after you have compiled a file or files. The linker produces an executable program file from your compiled files.

The library manager program, LIB.EXE, is used to create and organize libraries of object modules. You invoke this utility by typing "LIB".

EXEPACK.EXE and EXEMOD.EXE are special programs you can use to modify your executable program files. They are discussed in Sections 7.9.1 and 7.9.2, respectively, of Chapter 7, "Advanced Topics."

CL.EXE is an alternate control program for the compiler. It is provided for those users who are familiar with the cc command from XENIX or UNIX systems. Like MSC.EXE, CL.EXE invokes the four passes of the compiler for you. You can also invoke the linker through CL.EXE.

2.4.2 Include Files

Include files are text files you can incorporate into your program by using the C preprocessor directive **#include**. These files contain definitions used by run-time library routines.

By convention, some include files are stored in a subdirectory named SYS. This convention originated in the practice of storing files that define "system-level" constants and types in a separate "system" subdirectory on UNIX and XENIX systems. However, not all the include files that are traditionally stored in the SYS subdirectory contain system-level definitions, and some of the include files *not* in the SYS subdirectory contain system-level definitions. Since many programs, particularly those created under the XENIX and UNIX operating systems, rely on the SYS subdirectory convention, Microsoft continues to recognize this convention to maintain compatibility with existing programs.

2.4.3 Library Files

Library files contain compiled run-time library routines to be linked with your program. A separate set of library files is included for each standard memory model: small, medium, and large. The terms "small model," "medium model," and "large model" refer to standard memory models you can choose for your program, based on its storage requirements for code and data.

You do not have to choose a memory model in order to process and run your program. The small model is appropriate for most programs, and the compiler uses the small model and the small model library files by default.

Two additional library files, EM.LIB and 87.LIB, are model-independent; they can be used with all three memory models. EM.LIB is the floating-point emulator, used to perform floating-point operations. 87.LIB is the 8087/80287 floating-point library. This library provides minimal floating-point support and can only be used when an 8087 or 80287 coprocessor is present. The compiler uses the emulator (EM.LIB) by default, but you can override the default to use 87.LIB (if you have a coprocessor) or the alternate math library, described below. Floating-point options are described in more detail in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics."

The library files beginning with an "S" belong to the small model library set. SLIBC.LIB is the standard run-time library. SLIBC.LIB contains all the routines included in the Microsoft C run-time library except for math routines that require floating-point support.

SLIBC.LIB also contains an object module named CRT0.OBJ, which is the start-up routine for small model programs. The start-up routine performs several important tasks. It allocates the stack for your program and initializes the segment registers. It sets up the "argv," "argc," and "envp" variables to allow command-line arguments and environment settings to be passed to the program. The start-up routine is responsible for setting up and maintaining the operating environment for the program. The start-up routine also initializes the emulator, if loaded.

SLIBFP.LIB is the floating-point math library. It is required whenever your program uses EM.LIB or 87.LIB.

SLIBFA.LIB is the alternate floating-point library. You can use SLIBFA.LIB instead of EM.LIB and SLIBFP.LIB when speed is more important than precision in floating-point calculations. See the discussion of floating-point operations in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics," for details on this option.

When you compile a source file using MSC.EXE or CL.EXE, the compiler places the names of the standard library (SLIBC.LIB) and the floating-point libraries (EM.LIB and SLIBFP.LIB are the default) in the object file for the linker. Thus, LINK is able to link these libraries with your program automatically. If you compile using one of the /FP options, you can control which floating-point libraries are specified in the object files. You can also override the default at link time by substituting the name of a

different floating-point library for the library name in the object file. These options are discussed in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics."

The files beginning with an "M" are medium model library files, and the files beginning with "L" are large model library files. The organization and content of these files are analogous to that of the small model library set. MLIB.LIB and LLIBC.LIB, like SLIBC.LIB, each contain a start-up routine named CRT0.OBJ.

If you specify the medium or large model when you process your program, the compiler uses the appropriate standard library (MLIBC.LIB or LLIBC.LIB) and floating-point libraries (by default, EM.LIB plus MLIBFP.LIB or LLIBFP.LIB) when placing information in the object file for the linker. Otherwise, the compiler uses the small model files.

2.4.4 Other Files

The object file BINMODE.OBJ is provided for modifying the default mode for data files from text mode to binary mode. The same file can be used with all three memory models (see Section 7.10, "Controlling Binary and Text Modes," of Chapter 7, "Advanced Topics," for details on BINMODE.OBJ).

The SSETARGV.OBJ file provides a routine that expands the MS-DOS wild card characters "?" and "*" in filename arguments passed to C programs from the command line. SSETARGV.OBJ is the small model version of the routine; MSETARGV.OBJ and LSETARGV.OBJ are the medium and large model versions. Wild card expansion is performed only if you explicitly link with the appropriate SETARGV file. See Section 5.3, "Expanding Wild Card Arguments," in Chapter 5, "Running Your C Program," for details.

The README.DOC file, if present, contains documentation of recent changes that may not be included in this manual. If a README.DOC file is included on your disks, be sure to read the file before trying to use the software, since the file may contain information that affects how the compiler operates. In case of conflict between the manual and the README.DOC file, the README.DOC file takes precedence.

2.5 Setting Up the Environment

Before you compile and link a program using MSC.EXE and LINK.EXE, you must make sure that the programs can locate all the files they need to process your program. The required files are listed below.

Executable files	These are the files the control program executes as it processes your program. The names of these files are P0.EXE, P1.EXE, P2.EXE, and P3.EXE. When using CL.EXE, the alternate control program, LINK.EXE may also be executed by the control program. Notice that MSC.EXE and CL.EXE are also executable files.
Include files	If your program uses the preprocessor directive **#include**, the compiler attempts to find the given text file and include it in your program at compile time. Your program cannot be compiled if the given include file is not found.
Library files	At link time, LINK.EXE attempts to find the library files that are specified in the object file or on the link command line and link them with your program.

When you invoke the compiler or linker, it determines whether you have defined certain "standard places" to search for the necessary files. You can define these places by using environment variables. Environment variables are defined at the MS-DOS command level using the MS-DOS commands SET and PATH. They are called environment variables because they are effective throughout the environment in which a program is executed.

Although environment variables are usually helpful, you are not required to set them. If you do not set these variables, the current working directory is used to search for files and to create temporary files. An error is produced if the files are not found in the current working directory or if insufficient space is available in the directory for creating temporary files.

MSC.EXE looks for three environment variables: PATH, INCLUDE, and TMP. LINK.EXE uses one environment variable, LIB. The alternate control program, CL.EXE, uses all four environment variables.

PATH tells the compiler where to look for executable files, and INCLUDE tells it where to look for include files. The LIB environment variable tells LINK.EXE where to find any library files it needs.

The TMP environment variable has a slightly different function. The compiler creates a number of temporary files as it processes a program. The TMP environment variable tells the compiler where to create these files. The temporary files are removed by the time the compiler finishes processing. The space required for the temporary files is typically two times the size of the source file. It is often helpful to create the temporary files on another disk to avoid running out of space on your default disk.

Note

If you have a memory-based disk emulator (for example, the Microsoft RAMCard Memory Board), you can speed up processing by assigning it to the TMP variable.

To define the environment variables INCLUDE, LIB, and TMP, use the SET command to assign a directory specification or specifications to the variable. You must set PATH, INCLUDE, and TMP *before* invoking the compiler if you want the variables to be effective while the compiler is running. Similarly, you must set LIB before the linking stage.

The TMP variable can only be assigned one pathname. The INCLUDE and LIB variables can contain more than one pathname. Each pathname is separated from the next pathname by a semicolon (;). The compiler or linker searches through all directories specified, in order of their appearance, until it finds the file it needs. This means that include files and library files can be separated and placed in different directories.

For example, you can tell the compiler where to look for include files by setting the INCLUDE variable. Here is an example.

```
SET INCLUDE=B:\INCLUDE;B:\CUSTOM
```

The compiler will first look for include files on Drive B in the directory named INCLUDE; then, if necessary, the compiler will search the CUSTOM directory.

Use the PATH command instead of the SET command to define the PATH variable. (Although it is permissible to define the PATH variable with the SET command, using this method under versions of MS-DOS earlier than 3.0 can cause the PATH variable to work incorrectly for some directory specifications using lowercase letters.) To define the PATH variable using the PATH command, simply give the PATH command followed by a space (or an equal sign) and one or more directory specifications separated by semicolons. For example,

```
PATH A:\BIN;A:\LINKER
```

tells the compiler to search for executable files on Drive A in the directory named BIN, then, if necessary, in the LINKER directory.

MSC.EXE searches through all directories specified, in order of their appearance, until it finds the executable file it needs. Thus, executable files can be separated and placed in different directories, as long as the pathname of each directory containing an executable file appears in the PATH specification.

The MS-DOS operating system also uses the PATH setting to locate executable files. For example, when you invoke MSC.EXE (by typing "MSC"), the MS-DOS system locates MSC.EXE by looking in your default directory and in the directories specified in the PATH setting. If you include the pathname of the directory containing MSC.EXE (or CL.EXE) in your PATH setting, you can execute the control program from any directory.

Once you have set an environment variable, it remains effective until you reset it to a different value (or to an empty value) or until you turn off the machine. If you frequently set up your compiler files in a standard way, you should place SET and PATH commands in your AUTOEXEC.BAT file. Then you will be ready to use the compiler each time you boot your machine.

You can also use SET and PATH commands in an MS-DOS batch file to define the environment for a particular program or programs. If you frequently switch back and forth between different environments, you can save time by setting up batch files that contain the SET and PATH commands for each environment. Then you can just execute a batch file each time you want to switch to a new environment.

Certain command line options available with the compiler override the effect of environment variables. For example, the /X option (described in Section 3.5.6 of Chapter 3, "Compiling") tells the compiler not to automatically search the standard places for include files. The result is that the compiler does not search for include files in the directories specified by the INCLUDE variable.

2.6 Setting Up Your CONFIG.SYS File

Before you can run the compiler you must make sure that your CONFIG.SYS file allows the compiler to open at least 10 files. Check this by looking in your CONFIG.SYS file for the line

```
files=n
```

where *n* is some integer. If *n* is less than 10, edit CONFIG.SYS to set *n* to 10 (or greater). If you do not currently have a CONFIG.SYS file, create a file by that name and insert the following line.

```
files=10
```

It is recommended, but not required, that you also set the number of buffers allowed in your CONFIG.SYS file. Check your CONFIG.SYS for the line

```
buffers=n
```

where *n* is an integer. If *n* is not already set, 10 is a reasonable number.

After you have edited or created your CONFIG.SYS file, reboot the system so the new settings will take effect.

2.7 Using an 8087 or 80287 Coprocessor

If you have an 8087 or 80287 coprocessor, you should read Section 3.7, "Selecting Floating-Point Options," in Chapter 3, "Compiling." With an 8087 or 80287, you can perform fast, efficient floating-point operations. You may want to select one of the 8087 options described in Section 3.7.1, "If You Have an 8087 or 80287 Coprocessor," to take maximum advantage of your processor's capabilities.

2.8 Using an 80186, 80188, or 80286 Processor

You can use the compiler with an 80186, 80188, or 80286 processor without taking any special steps. However, to take advantage of your processor's capabilities you will probably want to use the /G1 or /G2 option when you compile your programs. These options enable the instruction set for the 80186/80188 and 80286 processors respectively (see Section 3.8 of Chapter 3, "Compiling").

2.9 Converting Existing C Programs

If you own the Microsoft C Compiler Version 2.03 or earlier, or if you have programs written for that compiler, turn to Appendix D, "Converting from Previous Versions of the Compiler," for a discussion of differences between this compiler and earlier versions.

2.10 Organizing Your Software

Before you begin using the compiler, you will probably want to spend some time organizing the files on your disks. The optimal arrangement of files depends on your specific needs and on how you most frequently use the compiler, as well as your machine configuration. You can also take advantage of the compiler's use of environment variables to determine search paths for various pieces of the software.

It is recommended that you create a separate directory for each type of file: executable, include, and library. (Refer to your MS-DOS documentation if you are unfamiliar with the procedures for setting up and maintaining directories.) The "system-level" include files are conventionally placed in a separate subdirectory of the include file directory named SYS, but this is not required.

If you use the SYS subdirectory convention, you should give the subdirectory name along with the filename when you use a "system-level" include file in your program. For example, the line

```
#include <sys\timeb.h>
```

causes the compiler to look for the subdirectory named "sys" in the directory specified by the INCLUDE variable, and to include the file *timeb.h* from that subdirectory. On the other hand, if you do not use the SYS convention, the line

```
#include <timeb.h>
```

is sufficient.

Notice that, although case is significant within C programs, case is not significant to MS-DOS. The names "sys" and "SYS" are equivalent when used as MS-DOS directory names.

Sample setups for hard disk systems and floppy disk systems are given below. It is not necessary to read both sections; just read the one that applies to your system.

2.10.1 Sample Hard Disk Setup

The following sample setup is suitable for a hard disk system. The setup includes only the small model library files. If all your programs are small model, or if you are not concerned with memory models at all, then the small model library files are the only ones you need. On the other hand, if you use more than one memory model in your programming, you will probably want to add the appropriate library files to the LIB directory.

The 8087/80287 floating-point library and the alternate math library are not included in the sample setup because you do not need both the regular floating-point library and the other floating-point libraries at the same time. If you want to use one of the other floating-point libraries, you can substitute it or add it to the LIB directory. Similarly, only the MSC.EXE control program is included in this setup. If you prefer to use CL.EXE, add it to the BIN directory or substitute it for MSC.EXE.

```
C:\BIN\MSC.EXE
C:\BIN\P0.EXE
C:\BIN\P1.EXE
C:\BIN\P2.EXE
C:\BIN\P3.EXE
C:\BIN\LINK.EXE
C:\BIN\LIB.EXE

C:\INCLUDE\ASSERT.H
C:\INCLUDE\CONIO.H
C:\INCLUDE\CTYPE.H
C:\INCLUDE\DIRECT.H
C:\INCLUDE\DOS.H
C:\INCLUDE\ERRNO.H
C:\INCLUDE\FCNTL.H
C:\INCLUDE\IO.H
C:\INCLUDE\MALLOC.H
C:\INCLUDE\MATH.H
C:\INCLUDE\MEMORY.H
C:\INCLUDE\PROCESS.H
C:\INCLUDE\SEARCH.H
C:\INCLUDE\SETJMP.H
C:\INCLUDE\SHARE.H
C:\INCLUDE\SIGNAL.H
C:\INCLUDE\STDIO.H
C:\INCLUDE\STDLIB.H
C:\INCLUDE\STRING.H
C:\INCLUDE\TIME.H
C:\INCLUDE\V2TOV3.H

C:\INCLUDE\SYS\LOCKING.H
```

```
C:\INCLUDE\SYS\STAT.H
C:\INCLUDE\SYS\TIMEB.H
C:\INCLUDE\SYS\TYPES.H
C:\INCLUDE\SYS\UTIME.H

C:\LIB\SLIBC.LIB
C:\LIB\SLIBFP.LIB
C:\LIB\EM.LIB
```

Using this setup, your environment variables can be given the values shown below.

```
PATH C:\BIN
SET INCLUDE=C:\INCLUDE
SET LIB=C:\LIB
SET TMP=C:\
```

Notice that the TMP setting simply specifies the root directory of Drive C. The temporary files created by the compiler are removed by the time processing is completed, so you don't need to create a separate directory to store them. MSC.EXE deletes the temporary files automatically; you are not responsible for removing them.

With this sample setup you can run the compiler and linker from any directory or disk. If you use the EXEPACK.EXE and EXEMOD.EXE utilities, put them in the BIN directory. Then you can execute the programs from any directory.

If you use one of the SETARGV files (SSETARGV, MSETARGV, or LSETARGV, depending on the memory model) to enable wild card expansion, or the BINMODE.OBJ file to change the default text processing mode, you can place the file either in your C program file directory or in the LIB directory. Notice, however, that the LIB environment variable is not used to find the SETARGV or BINMODE file; if it is not in your current working directory, you must specify a pathname at link time.

2.10.2 Sample Floppy Disk Setup

You will need at least two floppy disks to set up the files so that you can run the compiler. The sample setup given below uses two disks and assumes that you will swap these two disks in and out of Drive A as necessary. You can develop your programs and create listing files on a separate disk in Drive B.

This sample setup includes only the small model library files. You can save space by keeping only one set of library files on a disk, since any given program uses only one set (either the small, the medium, or the large model set). If all your programs are small model, or if you are not concerned with memory models at all, then the small model library files are the only ones you need.

The 8087/80287 floating-point library and the alternate math library are not included in this sample setup because you do not need both the regular floating-point library and the other floating-point libraries at the same time. If you want to use one of the other floating-point libraries, you can substitute it. Similarly, only the MSC.EXE control program is included in this setup. If you prefer to use CL.EXE instead, substitute it for MSC.EXE.

```
Disk 1:
        BIN\MSC.EXE
        BIN\P0.EXE
        BIN\P1.EXE
        BIN\P2.EXE
        BIN\P3.EXE

        INCLUDE\ASSERT.H
        INCLUDE\CONIO.H
        INCLUDE\CTYPE.H
        INCLUDE\DIRECT.H
        INCLUDE\DOS.H
        INCLUDE\ERRNO.H
        INCLUDE\FCNTL.H
        INCLUDE\IO.H
        INCLUDE\MALLOC.H
        INCLUDE\MATH.H
        INCLUDE\MEMORY.H
        INCLUDE\PROCESS.H
        INCLUDE\SEARCH.H
        INCLUDE\SETJMP.H
        INCLUDE\SHARE.H
        INCLUDE\SIGNAL.H
        INCLUDE\STDIO.H
        INCLUDE\STDLIB.H
        INCLUDE\STRING.H
        INCLUDE\TIME.H
        INCLUDE\V2TOV3.H

        INCLUDE\SYS\LOCKING.H
        INCLUDE\SYS\STAT.H
        INCLUDE\SYS\TIMEB.H
        INCLUDE\SYS\TYPES.H
```

```
        INCLUDE\SYS\UTIME.H

Disk 2:
        BIN\LINK.EXE
        BIN\LIB.EXE

        LIB\SLIBC.LIB
        LIB\SLIBFP.LIB
        LIB\EM.LIB
```

With this setup, all the files required in the compiling stage are on Disk 1, and all the files required in the linking stage are on Disk 2. This organization allows you to change disks easily when you have finished compiling a file and are ready to link.

Using this sample setup, your environment variables can be given the values shown below, assuming that Disks 1 and 2 will be swapped in and out of Drive A.

```
PATH A:\BIN
SET INCLUDE=A:\INCLUDE
SET LIB=A:\LIB
SET TMP=B:\
```

With this setup, you should create and store your programs on a separate disk on Drive B. You should also run the compiler from Drive B, so that B is the default drive for output files (the object file, listing file, map file, and executable program file). If you try to run the compiler from Drive A, there may be insufficient space available for creating these files. For the same reason, the TMP variable is assigned to Drive B.

Notice that the TMP setting simply specifies the root directory of Drive B. The temporary files created by the compiler are removed by the time processing is completed, so you don't need to create a separate subdirectory to store them. MSC.EXE deletes the temporary files automatically; you are not responsible for removing them.

If you use the EXEPACK.EXE and EXEMOD.EXE utilities, put them in the BIN directory on Disk 2. Then you can execute them from your program disk in Drive B.

If you use one of the SETARGV files (SSETARGV, MSETARGV, or LSETARGV, depending on the memory model) to enable wild card expansion, or the BINMODE.OBJ file to change the default text processing mode, you can place the file either in the directory with your C program files or in the LIB directory. Notice, however, that the LIB environment

variable is not used to find the SETARGV or BINMODE file; when it is not in your current working directory, you must specify a pathname at link time.

If you use more than one memory model in your programming, you will probably want to set up a separate library disk for each model. Notice that the files stored on Disk 1 (the compiler passes and the include files) do not change with the memory model, so you can use the same disk in the compiling stage for all three models.

On each separate library disk you will have the library files for that model, plus a copy of the LINK and LIB utilities. Although the LINK and LIB utilities do not change with the memory model, it is convenient to have a copy on each disk so that you can invoke LINK and LIB without changing back to your small model disk.

Use the same directory structure on all three disks (small, medium, and large). That way, you will not have to change the values of your environment variables when you change disks. For example, to process a medium model program using the alternate math library instead of the emulator, you could set up a disk like the following, to be used in Drive A.

```
BIN\LINK.EXE
BIN\LIB.EXE

LIB\MLIBC.LIB
LIB\MLIBFA.LIB
```

This organization is identical to the setup for Disk 2 given earlier, except that the medium model standard library file replaces the small model file, and the medium model alternate math library (MLIBFA.LIB) is used instead of EM.LIB and SLIBFP.LIB. The PATH setting (A:\BIN) and TMP setting (B:\) used above are valid for this disk as well, since it is organized with the same directory structure. Notice that you must use the same disk drive, Drive A, when you change from the small model disk to the medium model disk. Otherwise, your environment settings become invalid.

2.11 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft C Compiler. By following these steps you can produce and run an executable program file.

The source file used for this practice session is the sample source file DEMO.C, which is included with your compiler software. DEMO.C is a very simple C program that contains only one function, the *main* function. The *main* function is designed to print on your terminal any command line arguments you pass to the program at execution time. It will also print out the current value of environment settings. You can examine the DEMO.C source file to see how it accomplishes this. For a full discussion of passing command line data to programs, accessing the program environment from within a program, and declaring the "argc," "argv," and "envp" parameters, see Chapter 5, "Running Your C Program."

This practice session assumes that you are using the sample disk setup and environment that is appropriate for your system. The sample setup and compiler environment was discussed in Section 2.10, "Organizing Your Software."

The first thing you should do is verify that the compiler environment is set up correctly. You can do this by typing SET. When you give the SET command without an argument, it lists all environment variables and their current settings. Make sure the PATH, INCLUDE, TMP, and LIB variables are in the list and that they are set appropriately for your system, as shown below.

Hard Disk Settings	Floppy Disk Settings
`PATH=C:\BIN`	`PATH=A:\BIN`
`INCLUDE=C:\INCLUDE`	`INCLUDE=A:\INCLUDE`
`LIB=C:\LIB`	`LIB=A:\LIB`
`TMP=C:\`	`TMP=B:\`

If your settings do not match these, turn back to the previous section to review the disk setup and environment settings.

Once you have set up the environment, you are ready to begin processing DEMO.C using steps 1-13 below.

1. First, set up a directory to hold program files. The directory can be on the hard disk or on a floppy disk. You can give the directory any name you like; for this session, the name PROG will be used. Next, copy DEMO.C into the PROG directory.

2. Now you are ready to begin compiling. Make sure that the PROG directory is your current working directory (use the CD command to change directories if necessary.) Then give this command:

   ```
   MSC
   ```

   The MSC command invokes MSC.EXE, the compiler control program. MSC.EXE displays prompts on your screen to guide you through the compiling process.

3. The first message to appear on your screen is

   ```
   Microsoft C Compiler  Version 3.xx
   (C)Copyright Microsoft Corp 1984 1985
   Source filename [.C]:
   ```

   Following the "Source filename" prompt, specify the name of the file or files to be compiled. If you don't include the filename extension, MSC.EXE assumes that the extension is ".C" (or ".c"). Type

   ```
   DEMO
   ```

   in response to this prompt.

4. The next prompt is

   ```
   Object filename [DEMO.OBJ]:
   ```

   This prompt allows you to supply a name for the object file. Instead of typing a name, respond to this prompt by pressing the RETURN key, causing MSC.EXE to use the default response for the prompt. The default response for the "Object filename" prompt is to name the object file DEMO.OBJ. The object file is created in the current working directory, which is the PROG directory.

5. The next prompt is

   ```
   Object listing [NUL.COD]:
   ```

This prompt lets you create a listing of your object file, containing the machine instructions that correspond to your C instructions. Type

```
DEMO
```

in response to this prompt. MSC.EXE appends the default extension ".COD" and creates a listing named DEMO.COD. The listing file is created in the current working directory (PROG).

6. MSC.EXE now begins to compile your program. If your program has errors, they will be displayed as the compiler operates. (DEMO.C does not have errors.) When the compilation process is finished, the MS-DOS prompt reappears.

   You now have an object file named DEMO.OBJ and a listing file named DEMO.COD in your current working directory.

7. Next you need to link your program.

Important

If you are using a floppy disk setup, you should change the disk in Drive A at this point. Remove the disk containing the compiler files and include files, then insert the disk containing the LINK utility and the library files.

To link your file, simply type

```
LINK
```

The LINK command invokes the linker. You will see the following message on your screen.

```
Microsoft 8086 Object Linker
```

The message is followed by a version number and copyright notice.

8. The first linker prompt is

   ```
   Object Modules [.OBJ]:
   ```

   You have only one object file to link, so just type

```
DEMO
```

in response to this prompt. Microsoft LINK appends the ".OBJ" extension to find your file on the disk. Since the file is in the current working directory, you do not have to specify a pathname for LINK to find it.

9. The next prompt is

```
Run File [DEMO.EXE]:
```

This prompt lets you name the executable program file. Press the RETURN key in response to this prompt. The linker uses the default name shown in brackets for the executable file if you don't supply a different name. The executable file is created in the current working directory (PROG).

10. The next prompt is

```
List File [NUL.MAP]:
```

If you give a filename following this prompt, the linker creates a map file listing all the external symbols in your program and their locations. Type the response

```
DEMO /MAP
```

This response tells the linker to create a listing file named DEMO.MAP. The ".MAP" extension is used because you did not supply your own extension. The map file is created in the PROG directory by default. The /MAP option causes global symbols to be listed at the end of DEMO.MAP.

11. The final prompt is

```
Libraries [.LIB]:
```

The names of the standard C and floating-point libraries are provided in the object file, and the LIB environment variable tells the linker where to find the given library files. Therefore, you do not need to give any library names following this prompt. Just press the RETURN key.

12. Microsoft LINK now proceeds to link your file. If any errors are found, they are displayed on your screen. When the MS-DOS system prompt reappears, the linker has finished processing your file. You now have an executable file named DEMO.EXE in your

directory, plus an object listing named DEMO.MAP.

You may want to examine the object listing (DEMO.COD) and map file (DEMO.MAP) to familiarize yourself with their formats. These files are especially useful for debugging programs. However, the listing and map files are not required for running the program, so you can delete them if you like.

You can also delete the object file (DEMO.OBJ); since you have the executable program file, it is no longer needed. Chapter 6, "Managing Libraries," discusses how to use the Microsoft LIB program to organize object files into libraries of useful functions.

13. You can run the sample program simply by typing "DEMO". However, since the sample program is designed to take command line arguments and print them, you will probably want to give command line arguments when you run the program. For instance, you can run the program and pass three arguments by typing:

```
DEMO ONE TWO THREE
```

The program name is displayed on your screen, followed by the arguments ONE, TWO, and THREE and a listing of all current environment settings. The environment settings include PATH, LIB, INCLUDE, and TMP, as well as any other settings that are currently in effect (whether or not they apply to the C program or to the compilation and linking process).

Note

Under versions of MS-DOS earlier than 3.0, the program name is not available and will not be displayed.

This practice session used the simplest form of the MSC and LINK commands to show you their basic operation. The chapters that follow describe alternate forms and explain how to specify options with the MSC, LINK, and LIB commands. Notice that the CL command, described in Appendix C, "The CL Command," can be used to perform the same tasks as MSC and LINK.

2.12 Using Batch Files

You can create an MS-DOS batch file to set up the compiler environment and invoke the compiler. Creating and using batch files is discussed in full in your MS-DOS manual. This section is intended simply to demonstrate possible uses of the MSC command in a batch file.

A batch file is a text file containing a series of executable MS-DOS commands. Batch files always have the extension ".BAT". You execute a batch file by typing the filename without the ".BAT" extension. This causes MS-DOS to execute the series of commands the file contains.

Batch files are especially useful with the MSC command because they allow you to set up an environment before using the command. The examples below use the command line method of invoking MSC and Microsoft LINK. The command line method lets you give all responses to the prompts on a single line instead of waiting for the individual prompts. The command line method is discussed in Section 3.2.8 of Chapter 3, "Compiling," and in Section 4.4.10 of Chapter 4, "Linking."

For example, the following batch file, MYCOMP.BAT, could be used to create a program from a C source file in an environment set up for that purpose.

```
SET INCLUDE=B:\TOP\MYINC
MSC %1;
IF ERRORLEVEL 0 LINK %1,,%1;
```

The value given to INCLUDE in the first line alters the environment for the MSC command. Since no value is given for PATH, TMP, or LIB, their current values, if set, are unaffected by the batch file.

The symbol "%1" tells MS-DOS to look for an argument on the command line when you execute the batch file. When you type

```
MYCOMP THIS
```

the filename THIS is substituted for "%1", and THIS.C is compiled, producing the object file THIS.OBJ.

The line

```
IF ERRORLEVEL 0 LINK %1,,%1;
```

ensures that linking is only attempted if the source file was successfully compiled. The MSC and CL control programs return an exit code to allow testing for successful compilation. The exit code 0 indicates success; for information on the other codes, see Section 3.9.3, "Compiler Exit Codes," in Chapter 3, "Compiling." The MS-DOS batch command IF ERRORLEVEL is used to test the exit code; see your MS-DOS documentation for more on this command.

If compilation is successful, the object file THIS.OBJ is linked to produce THIS.EXE (the default name, since none is supplied). The name THIS is also supplied (by means of the symbol "%1") for the map file prompt, so a map file named THIS.MAP is produced.

Notice that the value given to INCLUDE when you execute the batch file remains in effect until you explicitly change it or until you reboot your machine. To restore your usual environment settings, you can create a batch file that resets the environment variables to the directories you most frequently use. For example, the following lines might be placed in a file called RESET.BAT, to be executed by typing RESET whenever you want to restore your usual environment settings.

```
PATH A:\HOME\BIN
SET INCLUDE=A:\INCLUDE
SET LIB=A:\LIB
SET TMP=B:\
```

Chapter 3
Compiling

3.1 Introduction 41
3.2 Running the Compiler 42
3.2.1 Filename Conventions 42
3.2.2 Special Filenames 43
3.2.3 Source Filename Prompt 44
3.2.4 Object Filename Prompt 44
3.2.5 Object Listing Prompt 44
3.2.6 Selecting Default Responses 45
3.2.7 Swapping Disks 45
3.2.8 Using the Command Line 46
3.2.9 Options 48
3.3 Naming the Object File 50
3.4 Producing Listing Files 52
3.5 Controlling the Preprocessor 54
3.5.1 Defining Constants and Macros 54
3.5.2 Predefined Identifiers 56
3.5.3 Removing Definitions of Predefined Identifiers 57
3.5.4 Producing a Preprocessed Listing 57
3.5.5 Preserving Comments 59
3.5.6 Searching for Include Files 59

3.6 Syntax Checking 60
3.6.1 Identifying Syntax Errors 61
3.6.2 Generating Function Declarations 61
3.7 Selecting Floating-Point Options 63
3.7.1 If You Have an 8087 or 80287 Coprocessor 64
3.7.2 If You Don't Have a Coprocessor 65
3.7.3 Compatibility Between Floating-Point Options 66
3.8 Using 80186, 80188, or 80286 Processors 68
3.9 Understanding Error Messages 68
3.9.1 C Compiler Messages 69
3.9.2 Setting the Warning Level 70
3.9.3 Compiler Exit Codes 72
3.10 Preparing for Debugging 72
3.11 Optimizing 73
3.12 Compiling Large Programs 75
3.13 Working with Memory Models 76
3.13.1 Creating Small Model Programs 79
3.13.2 Creating Medium Model Programs 79
3.13.3 Creating Large Model Programs 79

3.1 Introduction

One basic command, MSC, is all you need to compile your C source files with the Microsoft C Compiler. The MSC command takes care of executing the four compiler passes for you.

By drawing on the large set of MSC options, you can control and modify the tasks performed by the command. For example, you can direct MSC to create an object listing file or a preprocessed listing. Options also let you give information that applies to the compilation process, such as the definitions for manifest (symbolic) constants and macros and the kinds of warning messages you want to see.

The MSC command automatically optimizes your program. You never have to give an optimizing instruction, unless you want to change the way that MSC optimizes or you want to disable optimization altogether. See Section 3.11, "Optimizing," for more on these choices.

This chapter explains how to run the compiler using the MSC command and discusses commonly used MSC options in detail.

Additional MSC options are covered in Chapter 7, "Advanced Topics." A summary of the MSC command and all available options is provided in Section B.2 of Appendix B, "Command Summary," of this guide. Appendix C, "The CL Command," is a summary of the CL command, an alternative to the MSC command. The CL command is similar to the cc command on XENIX and UNIX systems, and is included for users who are comfortable with the XENIX cc command.

This chapter assumes that you know how to create, edit, and debug C program files on your system. For questions relating to the definition of the C language, see the *Microsoft C Language Reference.*

3.2 Running the Compiler

MSC requires two types of input: a command to start the compiler and responses to command prompts. Start the compiler by typing

```
MSC
```

at the MS-DOS command level. MSC prompts for the input it needs by displaying the following three messages, one at a time.

```
Source filename [.C]:
Object filename [.OBJ]:
Object listing  [NUL.COD]:
```

The responses you make to each prompt are explained below.

If you want to stop the compiling session for any reason, type CONTROL-C at any time. You will be returned to the MS-DOS command level, where you can restart MSC from the beginning.

3.2.1 Filename Conventions

You can use uppercase, lowercase, or a combination of both for the filenames you give in response to the prompts. For example, the following three filenames are considered equivalent.

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

You can include spaces before or after filenames, but not within them. Options (see Section 3.2.9) can appear anywhere spaces can appear.

MSC uses the default file extensions ".C", ".OBJ", and ".COD" when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells MSC that the specified file has *no* extension, while typing just "ABC" tells MSC to use the default extension for that prompt.

You can override any defaults by typing all or part of the name. For example, if the currently logged drive is B and you want the output file to be written to the disk in Drive A, type just the response "A:". The output file is written on Drive A with the default filename.

Notice that if you type any part of a legal pathname following the "Object listing" prompt, MSC produces a listing file. The default name is the "basename" of the source file with the extension ".COD". The basename of a file is the portion of the name preceding the period (.). For example, if you compile a file named TEST.C and type "A:" following the "Object listing" prompt, MSC produces a listing file on Drive A with the name A:TEST.COD.

3.2.2 Special Filenames

You can use the following MS-DOS device names as filenames with the MSC command. This allows you to direct files to your terminal or to a printer. Notice that you cannot use these names for ordinary filenames.

Name	Device
AUX	Refers to an auxiliary device (such as a printer or disk drive).
CON	Refers to the console (terminal).
PRN	Refers to the printer device.
NUL	Specifies a "null" (nonexistent) file. Giving NUL as a filename means that no file is created.

Even if you add device designations or filename extensions to these special filenames, they remain associated with the devices listed above. For example, A:CON.XXX still refers to the console and is not the name of a disk file.

Note

Object files contain machine code and are not printable. When responding to the "Object filename" prompt, do not give a filename that refers to a printer or console.

3.2.3 Source Filename Prompt

Following the "Source filename" prompt, give the name of the source file you want to compile. If you do not supply an extension, MSC automatically looks for a file with the ".C" extension.

Pathnames are allowed with the source filename. Therefore, you can specify the pathname of a source file in another directory or on another disk.

You may compile only one file at a time, so only one response to this prompt is allowed. There is no default response; MSC displays an error message if you do not supply a source filename.

3.2.4 Object Filename Prompt

Following the "Object filename" prompt, you can supply a name for the object file produced by compiling your source file. You are free to give any name and any extension you like. However, it is recommended that you use the conventional ".OBJ" extension because it simplifies operation of Microsoft LINK and Microsoft LIB, both of which use".OBJ" as the default extension when processing object files.

If you supply just a drive or directory specification following the "Object filename" prompt, MSC creates the object file in the given drive or directory and uses the default filename. You can use this option to create the object file in another directory or on another disk. When you give just a directory specification, the directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

The default name supplied for the object file is the basename of the source file with an ".OBJ" extension. If no pathname is supplied, the object file is created in the current working directory.

3.2.5 Object Listing Prompt

Following the "Object listing" prompt you can tell MSC to create an object listing for the compiled file. The object listing contains the machine instructions and assembled code for your program.

If you supply any filename following this prompt, MSC creates an object listing, using the filename you supply. By convention, these listings are given the extension ".COD", but you are free to choose any extension you like.

When you do not supply a filename, the default is the special name NUL.COD, which tells MSC *not* to create a listing.

The MSC command optimizes by default, so the object listing reflects the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear. To produce a listing without optimizing, use the /Od option, discussed in Section 3.10, "Preparing for Debugging."

To produce a combined source and assembly code listing, use the /Fc option, described in Section 3.4, "Producing Listing Files."

3.2.6 Selecting Default Responses

To select the default response to the current prompt, press the RETURN key without giving any other response. The next prompt will appear.

To select default responses to all remaining prompts, use a single semicolon (;) after the filename following the "Source filename" or "Object filename" prompt. Once the semicolon has been entered, you cannot respond to any of the remaining prompts for that compiling session. Any text appearing after the semicolon (such as an option) is ignored. Use the semicolon to save time when the default responses are acceptable.

There is no default for the first prompt, "Source filename". The default for the "Object filename" is the basename of the source file with an ".OBJ" extension. The default for the "Object listing" prompt is the special name NUL.COD, which tells MSC *not* to create an object listing file.

3.2.7 Swapping Disks

MSC suspends execution and displays a prompt whenever it cannot find one or more of the executable files that make up the compiler: P0.EXE, P1.EXE, P2.EXE, and P3.EXE. This behavior lets you store the compiler files on different disks if necessary and swap disks when MSC prompts you.

If you respond to the "Source filename" prompt with a nonexistent filename, or to the "Object filename" or "Object listing" prompts with an invalid pathname, MSC displays an error message and terminates. You must restart MSC with the correct information.

3.2.8 Using the Command Line

Once you understand how the MSC prompts and responses work, you can use the command line method of running the compiler. With this method you type all the filenames on the line used to start MSC. The command line method has the following form.

MSC *sourcename* [,[*objectname*]] [,[*listingname*]][;]

The entries following MSC are responses to the command prompts.

You can include spaces before or after filenames, but not within them. Options (described in Section 3.2.9) can appear anywhere spaces can appear.

You can leave the *objectname* and *listingname* fields blank to cause MSC to select the default responses. The semicolon (;) character has the same effect in the command line as it does with the MSC prompts. When MSC sees a semicolon on the command line, it uses the default responses to the remaining prompts. Any text after the semicolon on the command line is ignored.

The comma character serves as a separator and also has a special function in the command line. If you place a comma after the *objectname* field in the command line (whether or not an *objectname* is actually given), the default for the listing field is changed from NUL.COD to the basename of the source file plus ".COD". For example, the following two command lines are equivalent.

```
MSC TEST, TEST, TEST;
MSC TEST, , ;
```

In the first command line, the name TEST is explicitly specified for all three prompts, so TEST.C is compiled and two files are produced: TEST.OBJ and TEST.COD.

In the second command line, only the source filename is supplied. The default name (TEST.OBJ) is used for the object filename, since none is specified. The comma following the object filename field causes the default for the listing file to be changed to TEST.COD. Since no alternative name is supplied in the command line, a listing file named TEST.COD is created.

By contrast, the line

```
MSC TEST;
```

creates an object file named TEST.OBJ, but does not create a listing file, since no comma is present in the command line to change the default from NUL.COD to TEST.COD.

You can combine the prompt method and command line methods by giving MSC a partial command line. It prompts you for the fields you do not supply. You can end a partial command line with any of the items listed in the first column below. The second column describes the results.

semicolon (;)	MSC uses the default responses for the remaining prompts.
filename	MSC prompts you for the remaining responses, if any.
comma	If you give just a source filename followed by a comma, MSC prompts for both object filename and object listing name, as usual. However, if you supply both a source filename and an object filename, and then terminate the command line with a comma, MSC changes the default object listing name from NUL.COD to the basename of the source file plus ".COD". MSC then prompts you for an object listing name to allow you to override the default. (You can give the name NUL.COD to suppress the creation of an object listing.)

Options can also appear at the end of a partial command line, as discussed in the next section. The following examples demonstrate partial command lines.

Examples

```
1.  MSC ASK.C, TELL.OBJ
2.  MSC ASK, TELL;
3.  MSC ASK.C, TELL.OBJ,
4.  MSC ASK
```

Example 1 causes MSC to prompt with

```
Object listing[NUL.COD]
```

since you supplied the source filename and object filename but not the listing filename.

Notice the difference between Example 1 and Example 2, which tells MSC to use the default response (no file) for the object listing. No further prompts appear in this case.

In Example 3, the trailing comma (after TELL.OBJ) has a special meaning. It causes MSC to prompt as follows.

```
Object listing[TEST.COD]:
```

Notice that the default name in brackets is TEST.COD rather than NUL.COD. In this case an object listing is created by default, unless you override the default to specify a different listing name (or the name NUL.COD, to suppress the listing).

In Example 4, MSC starts prompting with the "Object filename" prompt, since only the source filename is supplied.

3.2.9 Options

The MSC command offers a large number of command options to control and modify the compiler's operation. Options begin with the forward slash character (/) and contain one or more letters. The hyphen character (-) can be used instead of the forward slash if you prefer. For example, /Zg and -Zg are both acceptable forms of the Zg option.

Important

Although filenames can be given in either uppercase or lowercase, *options must be given exactly as shown in this manual.* For example, /W and /w are two different options.

Options can appear anywhere a space can appear when you give the MSC command, except that options following a semicolon are ignored. Thus, options can go before or after any of the three filenames (source filename, object filename, and object listing.) The options apply to the entire compilation process, not just to the line on which they appear.

Some options take arguments, such as filenames, strings, or numbers. In most of these cases, spaces are allowed between the option letter and the argument. For example, these are both acceptable forms of the /W option:

```
/W 3
/W3
```

The /Gt option and /F family of options (/Fa, /Fc, /Fl, and /Fo, plus /Fe and /Fm with the CL command) form the only exceptions to this rule. The /Gt option accepts an optional numerical argument, while the /F options accept an optional pathname or partial pathname argument. When you supply an argument to one of these options, no spaces may appear between the option and the argument. For example,

```
/FcMINGLE
```

is acceptable, but

```
/Fc MINGLE
```

is not.

Some options consist of more than one letter. For example, the /F options mentioned above are two-letter options. No spaces are allowed between the letters of an option. Thus, in the above example, no spaces can appear between "F" and "c".

The order of the options is not important, and they can be given following any prompt or in any command line field. The default for the prompt is still used if you supply an option but no filename in response to the prompt.

The compiler options and the tasks they perform are discussed in the remainder of this chapter and in Chapter 7, "Advanced Topics." The command line form of the MSC command is used for the examples of options in this manual. Remember that you can use options with the prompts as well, as shown below.

Examples

```
1.  MSC
    Source filename [.C]:  A:\LOAD.C
    Object filename [LOAD.OBJ]: OUT
    Object listing  [NUL.COD]: /Oas /Fc

2.  MSC A:\LOAD.C /Oas /FoOUT /Fc;
```

The two examples above produce exactly the same effect. The source file LOAD.C on Drive A is compiled. The object file is named OUT.OBJ. The /Fc option produces a combined source and assembly code listing; since no argument was given with the /Fc option, the listing is given the default name LOAD.COD, formed by appending ".COD" to the basename of the source file. The object file and combined listing are both created on the default drive, since no drive was specified. The /Oas option tells the compiler how to optimize the object file. The Fc and Oas options are discussed in detail in Section 3.4, "Producing Listing Files," and Section 3.11, "Optimizing," respectively.

3.3 Naming the Object File

Option

/Fo*objectname*

You can name the object file produced by compiling your source file using the /Fo option. Using this option has the same effect as giving a filename at the "Object filename" prompt. When using the /Fo option, the *objectname* argument must appear immediately after the option, with no intervening spaces.

You are free to supply any name and any extension you like for the *objectname*. However, it is recommended that you use the conventional ".OBJ" extension because it simplifies operation of Microsoft LINK and Microsoft LIB, both of which use ".OBJ" as the default extension when processing object files. If you give an object filename without an extension, MSC automatically appends the ".OBJ" extension.

If you give just a drive or directory specification following the /Fo option, MSC creates the object file in the given drive or directory and uses the default filename (the basename of the source file plus ".OBJ"). You can use this option to create the object file in another directory or on another disk. When you give just a directory specification, the directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

If you give a name following the "Object filename" prompt and also use the /Fo option, the name you give after the /Fo option overrides the name you give following the prompt.

Examples

1. `MSC THIS, B:\OBJECT\;`

2. `MSC THIS /FoB:\OBJECT\;`

The two examples above produce exactly the same effect. The source file THIS.C is compiled; the resulting object file is named THIS.OBJ (by default). The directory specification "B:\OBJECT\" tells MSC to create THIS.OBJ in the given directory on Drive B.

3.4 Producing Listing Files

Options

/Fl [*listingname*]
/Fa [*listingname*]
/Fc [*listingname*]

You can create a listing of your compiled source file by responding to the "Object listing" prompt when you run MSC or by using the /Fl option. The object listing shows the machine instructions and assembled code for your program.

The /Fa listing produces an assembly listing of your program. The assembly listing contains the assembly code corresponding to your C file. The listing is suitable as input to the Microsoft Macro Assembler, also called Microsoft MASM.

To produce a listing that shows your source program along with the assembly code, use the /Fc option. This option produces a line-by-line combined source and assembly code listing, showing one line of your source program followed by the corresponding line (or lines) of machine instructions.

When using the /Fa, /Fc, and /Fl options, the *listingname,* if given, must follow the option immediately, with no intervening spaces. The *listingname* can be any one of the items listed in the first column below. The second column describes the results. If the *listingname* does not include an extension, the default extension is used. The default extension is ".COD" for the /Fc and /Fl options and ".ASM" for the /Fa option.

Filename	MSC uses the given filename, appending the default extension if the filename has no extension. The filename can include a path to tell MSC where to create the listing.
Directory specification	MSC creates the object listing in the given directory, using the default listing name, which is formed by appending the default extension to the basename of the source file. The directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

Omitted	When no *listingname* is given with the /Fl, /Fa, or /Fc option, MSC uses the default listing name (basename of the source file plus the default extension) and creates the listing in the current working directory.

The MSC command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the /Fc option to mingle the source and assembly code. To produce a listing without optimizing, use the /Od option (discussed in Section 3.11, "Optimizing") along with the listing option.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore. The Microsoft C compiler automatically prefixes an underscore to all global names to preserve compatibility with XENIX C compilers. If you write assembly language routines to interface with your C program, this naming convention is important; see Section 8.1.6, "Naming Conventions," in Chapter 8, "Interfaces with Other Languages."

In the listing file, you may also see names that begin with more than one underscore. Identifiers with more than one leading underscore are reserved for internal use by the compiler. You should not attempt to use these identifiers in your program. Moreover, you should avoid creating global names that begin with an underscore in your C source files. Since the compiler automatically adds another leading underscore, these names end up with two leading underscores, possibly causing conflicts with the names reserved by the compiler.

At most one listing file is produced each time you compile. The /Fc option overrides other listing options; whenever you use /Fc, a combined listing is produced.

Example

```
MSC HELLO.C, , /FcHELLO.LST;
```

The example creates a combined source and assembly code listing file named HELLO.LST and an object file named HELLO.OBJ from the source file HELLO.C.

3.5 Controlling the Preprocessor

The MSC command provides a number of options that give you control over the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source file listing. The options that perform these tasks are described below.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. See the *Microsoft C Language Reference* for a complete discussion of C preprocessor directives.

3.5.1 Defining Constants and Macros

Option

/D*identifier*[=[*string*]]

The /D option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and the *string* is its value or meaning.

If you leave out both the equal sign and the *string,* the given constant or macro is assumed to be defined, and its value is set to 1. For example, /DSET is sufficient to define SET.

If you give the equal sign with an empty string, the given constant or macro is considered defined; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, "/Dregister=" removes all occurrence of "register" from the source file. Notice that the identifier "register" is still considered to be defined.

The effect of using the /D option is the same as using a preprocessor **#define** directive at the beginning of your source file. The identifier is defined throughout the source file being compiled.

You can supply a command line definition for an identifier that is also defined within the source file. The command line definition holds up to the point of the redefinition in the source file.

Up to 16 definitions may appear on the command line, each preceded by the /D option. If you need to define more than 16 identifiers, see the discussion of the /U and /u options in Section 3.5.3, "Removing Definitions of Predefined Identifiers."

Example

```
MSC MAIN.C /D NEED=2;
```

The example defines the manifest constant NEED in the source file MAIN.C. Notice that spaces are permitted (but not required) between /D and the identifier. This definition is equivalent to placing the directive

```
#define NEED     2
```

at the top of the source file.

The /D option is especially useful with the **#if** directive. You can use the option to control compilation of statements in the source file. For example, suppose a source file named OTHER.C contains the following fragment.

```
#if defined(NEED)
   .
   .
   .
#endif
```

Suppose further that OTHER.C does not explicitly define NEED (that is,

no **#define** directive for NEED is present). Then all statements between the **#if** and the **#endif** directives are compiled only if you supply a definition of NEED by using /D. For instance, the command

```
MSC MAIN.C /DNEED;
```

is sufficient to compile all statements following the **#if** directive. Notice that NEED does not have to be set to a specific value to be considered defined. The following command, on the other hand, causes the statements in the **#if** block to be ignored (not compiled).

```
MSC MAIN.C;
```

3.5.2 Predefined Identifiers

The compiler defines four identifiers that are useful in writing portable programs. You can use these identifiers to compile code sections conditionally, depending on the current processor and operating system. The predefined identifiers and their functions are listed below.

Identifier	Function
MSDOS	Always defined. Identifies target operating system as MS-DOS.
M_I86	Always defined. Identifies target machine as a member of the I86 family.
M_I86*x*M	Always defined. Identifies memory model, where *x* is either S (small model), M (medium model), or L (large model). Small model is the default. Memory models are discussed later in this chapter.
SS_NE_DS	Defined only when memory model options are used to set up separate stack and data segments. Memory model options are discussed in Section 3.13, "Working with Memory Models," of this chapter and in Section 7.11, "Mixed Model Programming," of Chapter 7, "Advanced Topics."

3.5.3 Removing Definitions of Predefined Identifiers

Options

*/U**identifier***
/u

The /U (for "undefine") option can be used to turn off the definition of one or more of the predefined identifiers discussed in the previous section. The /u option turns off all four definitions.

These options are useful if you want to give more than 16 definitions on the command line, or if you have other uses for the predefined identifiers. For each definition of a predefined identifier you remove, you can substitute a definition of your own on the command line. When the definitions of all four predefined identifiers are removed, you can specify up to 20 command line definitions.

Example

```
MSC WORK /U MSDOS /U M_I86 /U M_I86SM;
```

This example removes the definitions of three predefined identifiers. Notice that the /U option must be given three times to do this.

3.5.4 Producing a Preprocessed Listing

Options

/P
/E
/EP

The /P, /E, and /EP options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation; no object file or listing is produced, even if you supply a name following the "Object filename" or "Object listing" prompt.

The /P option writes the preprocessed listing to a file with the same basename as the source file but with a ".I" extension.

The /E option copies the preprocessed listing to the standard output (usually your terminal), and places a **#line** directive in the output at the beginning and end of each included file. You can save this output by redirecting it to a file, using the MS-DOS redirection symbol ">" or ">>" (see your MS-DOS manual for a description of these symbols).

The /E option is useful when you want to resubmit the preprocessed listing for compilation. The **#line** directives renumber the lines of the preprocessed file so that errors generated in later stages of processing refer to the original source file rather than the preprocessed file.

Using the /EP option combines features of the /E and /P options: the file is preprocessed and copied to the standard output, but no **#line** directives are added.

Examples

```
1.  MSC MAIN.C /P;

2.  MSC ADD.C /E ;   > PREADD.C

3.  MSC ADD.C /EP ;
```

The first example creates the preprocessed file MAIN.I from the source file MAIN.C. The second command creates a preprocessed file with inserted **#line** directives from the source file ADD.C. The output is redirected to the file PREADD.C. The third command produces the same preprocessed output as the second example without the **#line** directives. The output appears on the screen.

3.5.5 Preserving Comments

Option

/C

Normally comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The /C (for "comment") option preserves comments during preprocessing. The /C option is valid only when the /E, /P, or /EP option is also used.

Example

```
MSC SAMPLE.C /P /C;
```

The example produces a listing named SAMPLE.I. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

3.5.6 Searching for Include Files

Options

/I *directory*
/X

The /I and /X options temporarily override or change the effects of the environment variable INCLUDE. These options let you give a particular file special handling without changing the compiler environment you normally use. (See Section 2.5, "Setting Up the Environment," in Chapter 2, "Getting Started," for a discussion of environment variables.)

You can add to the standard places for include files by using the /I (for "include") option. This option causes the compiler to search the directory you specify before searching the standard places given by the INCLUDE environment variable. You can add more than one include directory by giving the /I option more than once in the MSC command. The directories are searched in order of their appearance in the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs you must restart compilation with a corrected directory specification.

You can prevent the C preprocessor from searching the standard places for include files by using the /X (for "exclude") option. When MSC sees the /X option, it considers the list of standard places to be empty. This option is often used with the /I option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions. See the second example, below.

Examples

```
1.  MSC MAIN.C /I A:\INCLUDE /IB:\MY\INCLUDE;

2.  MSC MAIN.C /X /I B:\ALT\INCLUDE;
```

The first command directs the compiler to search for include files requested by MAIN.C first in the directory A:\INCLUDE, second in the directory B:\MY\INCLUDE, and finally in the directory or directories assigned to the INCLUDE environment variable.

In the second example, the compiler looks for include files only in the directory B:\ALT\INCLUDE. First the /X option tells MSC to consider the list of standard places empty; then the /I option specifies one directory to be searched.

3.6 Syntax Checking

The options described in this section are useful in the early stages of program development. They allow you to identify syntax errors and argument type mismatches quickly, without having to compile the source file.

3.6.1 Identifying Syntax Errors

Option

/Zs

The /Zs option causes the compiler to perform a syntax check only. No code is generated and no object file is produced. If the source file has syntax errors, error messages will be displayed.

This option provides a quick way to locate and correct syntax errors before attempting to compile a source file.

Example

```
MSC /Zs PRELIM.C;
```

This command causes the compiler to perform a syntax check on PRELIM.C, displaying messages about any errors it finds.

3.6.2 Generating Function Declarations

Option

/Zg

The /Zg option generates a function declaration for each function defined in the source file. The function declaration includes the function return type and an argument type list created from the types of the formal parameters of the function. Any function declarations already present in the source file are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using the MS-DOS redirection symbol ">" or ">>".

When the /Zg option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type-checking. The presence of a declared argument type list for a function "turns on" the compiler's type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type-checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type-checking is a recent addition to the C language, so existing C programs do not have argument type lists. See the *Microsoft C Language Reference* for details on function declarations and argument type lists.

You can use the /Zg option even if your source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not conflict. No conflict occurs when one declaration has an argument type list and another declaration of the same function does not, as long as the declarations are identical otherwise.

Your program may include calls to Microsoft C run-time library routines. The include files provided with the Microsoft C run-time library contain function declarations so that you can enable type-checking on library calls. The declarations are enclosed in preprocessor **#ifdef** blocks and are included only if you define the special identifier LINT_ARGS. You can define LINT_ARGS either with a **#define** directive in your program or by using the /D option when you compile.

Example

```
MSC FILE.C /Zg;
```

The above command causes the compiler to generate argument type lists for functions defined in FILE.C.

3.7 Selecting Floating-Point Options

Options

`/FPa`	Generates floating-point calls and selects alternate math library
`/FPc`	Generates floating-point calls and selects emulator library
`/FPc87`	Generates floating-point calls and selects 8087/80287 library
`/FPi`	Generates in-line instructions and selects emulator library
`/FPi87`	Generates in-line instructions and selects 8087/80287 library

The Microsoft C compiler offers several methods of handling floating-point operations. This section provides an overview of the floating-point options available and discusses the default floating-point behavior. For more detailed information on the floating-point libraries, plus a discussion of overriding floating-point options at link time and using the NO87 environment variable, see Section 7.7, "Controlling Floating-Point Operations," in Chapter 7, "Advanced Topics."

The Microsoft C Compiler can use an 8087 or 80287 coprocessor if one is present and can emulate 8087 operation through the use of an emulator library if not. The emulator library (EM.LIB) provides a large subset of the functions of an 8087/80287 in software. The emulator can perform basic operations to the same degree of accuracy as an 8087/80287. However, the emulator routines used for transcendental math functions differ slightly from the corresponding 8087/80287 functions, causing a slight difference (usually within 2 bits) in the results of these operations when performed with the emulator instead of with an 8087/80287.

By default, the Microsoft C compiler handles floating-point operations by making calls to the emulator library (this is the /FPc option). The emulator library is loaded, but if an 8087 or 80287 coprocessor is present at run time, the coprocessor will be used instead of the emulator. This method of handling floating-point operations always works, whether or not you have a coprocessor installed. Thus, you do not have to give a floating-point option at compile time unless you want to use one of the other options described below.

When you compile a source file using one of the floating-point options, the name of the required floating-point library (or libraries) is placed in the object file. At link time, the linker refers to the names in the object file to link with the appropriate libraries. You can override the library name given in the object file at link time and link with a different library instead; see Section 7.7.1, "Changing Libraries at Link Time," in Chapter

7, "Advanced Topics," for details. The only restriction on overriding at link time is that you are not allowed to change to the alternate math library after you have compiled using the /FPi or /FPi87 option.

3.7.1 If You Have an 8087 or 80287 Coprocessor

The /FPi87 option is the fastest and smallest option available for floating-point operations. It generates in-line instructions for an 8087/80287 coprocessor and selects the 8087/80287 library (87.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. An 8087 or 80287 *must* be present at run-time if the /FPi87 option is used.

The /FPc87 option generates function calls to routines in the 8087/80287 library (87.LIB) that perform the corresponding 8087/80287 instructions. The 8087/80287 library (87.LIB) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model, are selected. The /FPc87 option is slower than /FPi87 because it makes function calls instead of using in-line instructions. However, /FPc87 is more flexible. Using the /FPc87 option allows you to change your mind at link time (without recompiling the file) and use either the emulator or the alternate math library instead of relying on an 8087/80287 coprocessor. This is possible because the calls to 8087/80287 instructions are interchangeable with calls to the emulator and the alternate math library. See Section 7.7.1 of Chapter 7, "Advanced Topics," for instructions on changing libraries at link time.

Both the /FPi87 and /FPc87 options select the 8087/80287 library (87.LIB), which provides minimal floating-point support. Whenever 87.LIB is used, an 8087 or 80287 coprocessor must be present at run time. If no coprocessor is present, the program will not run and the message

```
Floating point not loaded
```

will appear.

The /FPi option generates in-line instructions for an 8087/80287 and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. If an 8087/80287 coprocessor is present at run time, it will be used. If not, the emulator is used.

Loading the emulator requires approximately 7K of additional space, so programs that use the /FPi option are larger than programs that use /FPi87. However, /FPi is a particularly useful option when you do not know in advance whether an 8087 or 80287 coprocessor will be available at run time.

In some cases, you may not want to use an 8087 or 80287 coprocessor, even though one is present. For example, you may be developing programs to run on systems that lack coprocessors. Conversely, you may want to write programs that can take advantage of an 8087/80287 at run time, even though you don't have one installed. There are several ways to control the use of an 8087 or 80287:

1. Use the /FPc (default) or /FPi option to specify use of an 8087/80287 if present, and use of the emulator if not. To use the emulator even when an 8087 or 80287 is present, set the NO87 environment variable, as discussed in Section 7.7.2 of Chapter 7, "Advanced Topics."
2. Use the /FPc87 or /FPi87 option if you always want to use a coprocessor. Programs compiled with these options will fail if a coprocessor is not present at run time.

3.7.2 If You Don't Have a Coprocessor

The /FPi option generates in-line instructions for an 8087/80287 coprocessor and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. If an 8087/80287 is present at run time, it will be used. If not, the emulator library, which mimics the operation of an 8087, is used. Because this option uses in-line instructions, it is the most efficient way to get maximum precision in floating-point operations without a coprocessor.

The /FPc option is the default when you do not specify a floating-point option. It generates floating-point calls to the emulator library and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. The /FPc option is slower than /FPi because it makes function calls instead of using in-line instructions. However, /FPc is more flexible than /FPi. Using the /FPc option allows you to change your mind at link time (without recompiling the file) and use an 8087/80287 coprocessor or the alternate math library instead of using the emulator. This is possible because the same function call interface is provided in all three libraries: the 8087/80287 library, the alternate math library, and the emulator library. See Section 7.7.1 of

Chapter 7, "Advanced Topics," for instructions on changing libraries at link time.

The /FPa option generates floating-point calls and and selects the alternate math library (SLIBFA.LIB, MLIBFA.LIB, or LLIBFA.LIB, depending on the memory model). The alternate math library uses a subset of the IEEE (Institute of Electrical and Electronics Engineers, Inc.) standard format numbers, sacrificing some accuracy for speed and simplicity. (Infinities, NaN's, and denormal numbers are not used.) Calls to this library provide your fastest and smallest option if you do not have an 8087 or 80287 coprocessor. With this option, as with the /FPc option, you can change your mind at link time and use the emulator or an 8087/80287 instead; see Section 7.7.1 of Chapter 7, "Advanced Topics," for details.

In some cases, you may want to write programs that will be able to take advantage of an 8087 or 80287 at run time, even though you don't have one installed. See Section 3.7.1, "If You Have an 8087 or 80287 Coprocessor," for a description of the appropriate options.

3.7.3 Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link more than one source file together to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way and that the environment is set up properly to allow the linker to find the required libraries. See Chapter 4, "Linking," for a detailed discussion of linking.

Note

If you are building libraries of C routines that contain floating-point operations, the /FPc (default) floating-point option is recommended for all compilations. The /FPc option offers the greatest amount of flexibility.

Whenever a file is compiled using the /FPi or /FPi87 option, in-line instructions are generated. In the case of the /FPi87 option, the library file 87.LIB and either SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model, must be present at link time, and an

8087/80287 coprocessor must be present at run time. For /FPi, the emulator library (EM.LIB) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB must be present at link time, and either the emulator or an 8087/80287 must be present at run time. As long as these requirements are satisfied, object files produced using the /FPi and /FPi87 options can be linked together without compatibility problems. Such object files can also be linked with object files produced using /FPa, /FPc, or /FPc87.

Whenever a file is compiled with the /FPa, /FPc, or /FPc87 options, floating-point function calls are generated. Each option places the name of the appropriate library file or files in the object file. However, when linking several such object files together, you must be aware of the process used to resolve the function calls.

Since floating-point calls to the emulator, the alternate math library, and 8087/80287 coprocessor instructions are interchangeable, only one library is used at link time to resolve the calls. In other words, you must choose one of these libraries per program; the same program cannot make calls to more than one library.

You can control which library is used in one of two ways:

1. At link time, as the *first* name in the list of object files to be linked, give an object file that contains the name of the desired library. For example, if you want to use the alternate math library, give the name of an object file compiled using the /FPa option. All floating-point calls will refer to the alternate math library.
2. At link time, give the /NOD (no default library search) option and then give the name of the floating-point library file or files you want to use in the "Libraries" field. This library overrides the names in the object files, and all floating-point calls will refer to the named library. Since the /NOD option causes all default libraries to be ignored, you must also specify the name of the standard C library (SLIBC.LIB, MLIBC.LIB, or LLIBC.LIB). Always give the names of the floating-point libraries *before* the name of the standard C library in the "Libraries" field.

3.8 Using 80186, 80188, or 80286 Processors

Options

```
/G0
/G1
/G2
```

If you have an 80186, 80188, or 80286 processor, you can use the /G1 or /G2 option to enable the instruction set for your processor. Use /G1 for 80186 and 80188 processors; use /G2 for an 80286. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80286 instruction set.

The /G0 option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly since the 8086/8088 instruction set is used by default.

3.9 Understanding Error Messages

The C compiler generates a broad range of error and warning messages to help you locate errors and potential problems in programs. The following sections describe the form and meaning of the compiler error messages and warning messages you can encounter while using the MSC command. For a list of actual error messages, see Appendix E, "Error Messages."

Error messages produced by the compiler are sent to the standard output, which is usually your console. You can redirect the messages to a file or printer by using an MS-DOS redirection symbol, ">" or ">>". This is especially useful in batch file processing. For example, the following command redirects error messages to the printer device (designated by PRN).

```
MSC ALPHA.C; > PRN
```

Note that only output that ordinarily goes to the console screen is redirected. The object file is given the name ALPHA.OBJ and is created in the current working directory.

3.9.1 C Compiler Messages

The C compiler displays messages about syntactic and semantic errors, such as misplaced punctuation, illegal use of operators, and undeclared variables, in a source file. It also displays warning messages about statements containing potential problems caused by data conversions or the mismatch of types. If you give invalid or incompatible command line options, the compiler will notify you of the error.

The error messages produced by the C compiler fall into five categories: warning messages, fatal error messages, compilation error messages, command line messages, and compiler internal error messages.

Warning messages are informational only; they do not prevent compilation and linking. These messages alert you to potential problems such as type mismatches, data conversions, redeclarations, and overflow conditions. The conditions described by warning messages are not necessarily illegal or undesirable, but you should examine the messages carefully to verify that your program produces these conditions intentionally. Otherwise, your program may not operate as you expect. You can control the level of warnings generated by the compiler by using the /W option, as described in Section 3.9.2.

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program. Fatal errors can be caused by problems such as insufficient disk space or malformed preprocessor commands. After printing out a message about the fatal error, the compiler terminates without producing an object file or checking for further errors.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. When the compiler encounters a nonfatal program error, it attempts to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler terminates processing.

Command line messages give you information about invalid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler terminates processing.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. See Section E.3, "Compiler Error Messages," in Appendix E, "Error Messages," for instructions on how to notify Microsoft about internal compiler errors.

Error messages of all types have the same basic form:

filename (*linenumber*) : *message*

where *filename* is the name of the source file being compiled, *linenumber* identifies the line of the file containing the error, and *message* is a self-explanatory description of the error or warning. For warning messages and fatal errors, the word "warning" or "fatal" appears at the beginning of the message, followed by a colon.

The messages for each category are listed in alphabetical order and described in more detail in Appendix E, "Error Messages."

3.9.2 Setting the Warning Level

Option

/W number
/w

You can set the level of warning messages produced by the compiler by using the /W (for "warning") option. This option directs the compiler to display messages about statements that may not be compiled as the programmer intends. Warnings indicate potential problems rather than actual errors.

To use the /W option, choose one of the warning levels described in Table 3.1 and specify the corresponding *number* after the option. The /w option provides a shorter way to say /W 0 and has the same effect.

Table 3.1
Warning Levels

Level	Warning
0	Suppresses all warning messages. Only messages about actual syntactic or semantic errors are displayed.
1	Warns about potentially missing statements, unsafe conversions, and other structural problems. Also, warns about overt type mismatches.
2	Warns about all type mismatches (strong typing).
3	Warns on all automatic data conversions.

The default is level 1, so you do not need to give the /W option when you want level 1.

The higher option levels are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels are best for compiling programs whose questionable statements are intentionally designed.

Example

```
1.  MSC /W 3 MAIN.C;

2.  MSC /w MAIN.C;
```

The first command directs the compiler to perform the highest level of checking, and produces the greatest number of warning messages. The second command causes MAIN.C to be compiled at the lowest level of checking, with no warning messages. Note that the /w option has the same effect as /W 0.

3.9.3 Compiler Exit Codes

The MSC control program returns an exit code of 0, 2, or 4 to indicate the status of the compilation. The exit code is useful with the MS-DOS batch command IF ERRORLEVEL; it allows you to test for the success or failure of the compilation before proceeding with other tasks in the batch file. The exit codes listed in the first column below have the meanings described in the second column.

0	Successful compilation. Notice that compilation can be successful even if warning messages are produced.
2	Unsuccessful due to program errors.
4	Unsuccessful due to system-level errors (such as insufficient disk space) or compiler internal errors. Compiler internal errors indicate an error on the part of the compiler rather that your program.

See Appendix E, "Error Messages," for details about specific error messages.

3.10 Preparing for Debugging

Options

```
/Zd
/Od
```

The /Zd option produces an object file containing line number records that correspond to the line numbers of the source file. The /Zd option is useful when you want to pass an object file to a symbolic debugger. The debugger can use the line numbers to refer to program locations. (See Section F.5 of Appendix F, "Working with Microsoft Products," for information on using SYMDEB, the symbolic debugger for Microsoft Macro Assembler, with C programs.)

The /Od option tells the compiler *not* to perform optimization. Without the /Od option, the default is to optimize. You may want to use this option when you plan to use a symbolic debugger with your object file. Optimization can involve rearrangement of instructions. If you optimize before debugging, it may be difficult to recognize and correct your code.

Other optimization options are discussed in Section 3.11, "Optimizing."

Example

```
MSC TEST.C, /Zd /Od, TEST.COD;
```

This command produces an object file named TEST.OBJ that contains line numbers corresponding to the line numbers of TEST.C. A listing file, TEST.COD, is also created. No optimization is performed.

3.11 Optimizing

Option

/Ostring

The optimizing procedures available with the Microsoft C Compiler can reduce the storage space and execution time required for a compiled program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimization by default. You can use the /O (for "optimize") options to exercise greater control over the optimizations performed. Some additional advanced optimizing procedures are discussed in Section 7.8 of Chapter 7, "Advanced Topics."

The *string* after the /O option lets you influence how the compiler performs optimization. The string is formed from the following characters.

Character	Optimizing Procedure
`s`	Favor code size during optimization
`t`	Favor execution time during optimization
`d`	Disable optimization
`a`	Relax alias checking

The letters can appear in any order: /Oat and /Ota have the same effect. The letter "x" is also available with the /O option to perform maximum optimization, as discussed in Section 7.8.2 of Chapter 7, "Advanced Topics."

When you do not give an /O option to the MSC command, it automatically uses /Os, meaning that code size is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps less efficient) and larger (but perhaps more efficient) code, the compiler chooses to generate smaller code. To cause the compiler to favor execution time instead (generating more efficient but perhaps larger code), use the /Ot option.

The /Od option turns off optimization. This option is useful in the early stages of program development to avoid optimizing code that will be changed. Because optimization may involve rearrangement of instructions, you may also want to specify the /Od option when you use a debugger with your program or when you want to examine an object file listing. If you optimize before debugging, it can be difficult to recognize and correct your code.

The "a" option can be used with either the "s" or the "t" option to relax alias checking. The compiler performs alias checking to make sure that it does not eliminate instructions incorrectly when you refer to the same memory location by more than one name. You should include the "a" option only when you are sure that your program does not use aliases.

For example, consider the following code fragment.

```
int count, *pc;
pc = &count;
count = 0;
   .
   .
   .
(*pc)++;
   .
   .
   .
count = 0;
```

The reference to *count* through a pointer, (**pc*), is known as an "alias" for *count* because it provides another way to access the same memory location. When the compiler performs alias checking, it detects the indirect reference to *count* through *pc* and does *not* eliminate the second instruction that assigns zero to *count*.

When you use the "a" option, you are telling the compiler that your program does not use aliases. Therefore, the compiler does not check for indirect references, such as the reference to *count* through a pointer. It would be an error to use the "a" option with the above example. The

compiler would see only that the same value, 0, is assigned to *count* twice, without any intervening assignments that change its value. The second assignment would be considered redundant and would be eliminated in the optimization stage, possibly causing the program to give incorrect results.

Example

```
MSC FILE.C /Ota;
```

This command tells the compiler to relax alias checking and to optimize for faster execution time when it compiles FILE.C.

3.12 Compiling Large Programs

If you are compiling a program or file with more than 64K (kilobytes) of data or with more than 64K of code you should use one of the memory models described in Section 3.13, "Working with Memory Models." (A kilobyte is 1,024 bytes.) You can use map files to determine data and code sizes for each individual program file.

The compiler uses a small memory model by default. The small memory model allocates one segment each, up to 64K in size, for the code and data of your program. (The code segment of a program may also be referred to as the "text" segment.) MSC produces an error message, like the one listed below, if an individual file exceeds these limits.

```
Illegal allocation of segment-type segment > 64K
```

The *segment-type* will specify either "data" or "code," depending on which limit was exceeded.

Even if no individual file exceeds the small model restrictions, you may exceed the 64K limit when you link several compiled files together to form a large program. If this occurs you must recompile the files using a larger memory model. Using a medium memory model allows you to create programs with more than 64K of code (the 64K-restriction on data still applies, however). In large model programs, code and data can both exceed 64K (although no single data item can be larger than 64K).

If your program exceeds the 64K limit on data but has less than 64K of code, you may want to use the **far** keyword for one or more data items. See Section 7.11.1 of Chapter 7, "Advanced Topics," for a discussion of this option.

Even in medium and large model programs you cannot exceed the limit of 64K of code per program file compiled. The total code size for the program may be greater than 64K, but each individual program file (or "compiland") must contain less than 64K of code.

3.13 Working with Memory Models

Options

/Aletter
/Astring

The MSC command lets you create programs of a variety of sizes and purposes using the /A options. The /A option has two forms. In the first form, you give a single capital letter after /A to select one of the three standard memory models (small, medium, and large) defined by MSC. The compiler uses the small model by default. If your program is too large to compile with the default small model, use a medium or large model, as described below.

In the second form, you supply a string of three lowercase letters that tells the compiler the attributes of the memory model you want to use. The second form is more flexible but it requires a thorough understanding of the memory model attributes: code pointer size, data pointer size, and the stack and data segment setup.

The remainder of this chapter discusses the first form of the memory model option (/A*letter*). For a discussion of the second form of the memory model option (/A*string*) and the **near** and **far** keywords, see Section 7.11, "Mixed Model Programming," in Chapter 7, "Advanced Topics."

The memory model options allow you to set up memory in the optimal way for your program. They also determine how the system loads the program for execution.

C programs in memory consist of the actual machine code created from the program's source statements and the data storage created for the program's variables. The data storage, in most cases, also contains the stack used by the program for temporary storage during execution.

The MS-DOS system loads the code and data into "segments" in physical memory. Each segment is up to 64K bytes long. Separate segments are always allocated for the program code and data, so the minimum number of segments allocated for a program is two. Depending on its size (and the use of **near** and **far**, as discussed in Section 7.11.1 of Chapter 7, "Advanced Topics"), a program may require more segments for its code or data. Note that the program size includes any data and code required for library routines.

Three commonly used memory models are defined for you by the MSC command: the small model, the medium model, and the large model. Library support is provided for each of these standard models. Each model defines a different type of program structure and storage.

Small model programs are typically C programs that are short or have a limited purpose. Code and data for these programs each occupy one segment, so they are limited to 64K bytes each (128K bytes maximum total).

Medium model programs are typically C programs that have a large number of program statements but a relatively small amount of data. Program code can occupy any amount of space and is given as many segments as needed. However, no single module (program file) can exceed 64K bytes of code. Program data must not exceed 64K bytes total.

Large model programs are typically very large C programs that use a large amount of data storage during normal processing. Program code can occupy any amount of space, as long as no single module exceeds 64K bytes. Program data may have any size, except that the program must not contain any single data item exceeding 64K bytes.

The limitation on data item size in the large model program allows the compiler to perform address arithmetic on just 16 bits (the offset portion) of the address to refer to individual elements or members of the item. This is much more efficient than using a full 32-bit address and is possible because all elements or members of an item are known to reside in the same segment.

When you choose one of these memory models, the compiler operates with certain assumptions about the addresses of code and data for your program. In a small model program, all code is stored in a single segment and all data are stored in a single segment. Because the segment addresses are constant for all code items and all data items, the segment address is not required each time an item is addressed. Instead, any items in the program can be addressed with just an offset from the appropriate segment address. Only 16 bits are required to store an offset from an address, as opposed to 32 bits for a full segmented address. Thus, the compiler generates 16-bit, or "near", pointers for use in small model programs. This is the smallest and fastest option.

A medium model program uses multiple segments for code and a single segment for data. The address of a function, for example, in a medium model program must include the address of the appropriate code segment plus the offset of the beginning of the function from the base of that segment. Full 32-bit, or "far" pointers, are generated by the compiler to access code items in a medium model program. However, just an offset is sufficient for data items, since all data reside in one segment. Data items are accessed with "near" pointers in a medium model program. The medium model provides a useful trade-off of speed and space, since most programs refer more frequently to data items than to code.

A large model program requires the compiler to produce "far" pointers for both code and data items, since multiple segments are allotted for both code and data. Although this is the slowest option, the large model is useful because it can accommodate very large programs.

To provide additional flexibility within the standard memory models, the Microsoft C Compiler allows you to override the default addressing conventions for individual program items by using the special **near** and **far** keywords. These keywords let you access an item with either a "near" or a "far" pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program.

3.13.1 Creating Small Model Programs

Option

/AS

The small model option tells the compiler to create a program that occupies two segments, one each for code and data, when loaded into physical memory.

The compiler creates small model programs by default when you do not otherwise specify a program model. Thus, the /AS option is never required.

3.13.2 Creating Medium Model Programs

Option

/AM

The medium model option creates one segment for the data of the program and one code segment per file compiled.

3.13.3 Creating Large Model Programs

Option

/AL

The large model option directs the compiler to create multiple segments, as needed, for both instructions and data. However, no single module can exceed 64K bytes of code, and no single data item can exceed 64K bytes.

Chapter 4
Linking

4.1 Introduction 83
4.2 How the Linker Works 83
4.3 Linking C Program Files 84
4.3.1 The Main Function 85
4.3.2 Default Libraries and the Library Search Path 85
4.3.3 Changing the Default Libraries 86
4.3.4 LINK Options to Avoid 86
4.4 Running the Linker 87
4.4.1 Filename Conventions 88
4.4.2 Object Modules Prompt 88
4.4.3 Run File Prompt 89
4.4.4 List File Prompt 89
4.4.5 Libraries Prompt 89
4.4.6 Separating Entries 91
4.4.7 Extending Lines 91
4.4.8 Selecting Default Responses 91
4.4.9 Terminating the Link Session 92
4.4.10 Using the Command Line 92
4.4.11 Using a Response File 93

4.5 Controlling the Linker 94
4.5.1 Pausing During Linking 95
4.5.2 Producing a Listing File 96
4.5.2.1 Listing Public Symbols 97
4.5.2.2 Displaying Line Numbers 99
4.5.3 Ignoring Case 99
4.5.4 Ignoring Default Libraries 100
4.5.5 Controlling Stack Size 101
4.5.6 Allocating Paragraph Space 102
4.5.7 Controlling Segments 102
4.5.8 Using Overlays 103
4.5.8.1 Restrictions 104
4.5.8.2 Overlay Manager Prompts 104
4.5.9 Setting the Overlay Interrupt 105
4.5.10 Ordering Segments 105
4.5.11 Controlling Data Loading 106
4.5.12 Controlling Run File Loading 106
4.5.13 Preserving Compatibility 107

4.1 Introduction

The Microsoft LINK Object Code Linker links object files compiled on 8086/8088 machines and produces an executable run file as output. The format of input to Microsoft LINK is a subset of the Intel® object module format standard.

The output file from Microsoft LINK (the run file) is not bound to specific memory addresses. It can, therefore, be loaded and executed by the operating system at any convenient address. Microsoft LINK can produce executable files containing up to 1 megabyte of code and data.

4.2 How the Linker Works

Microsoft LINK performs the following steps to combine object modules and produce a run file.

1. Reads the object modules you submit
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads data in the segments
6. Reads all relocation references in object modules
7. Performs fix-ups
8. Outputs a run file (executable image) and relocation information

Microsoft LINK produces a list file that shows how external references are resolved and prints any error messages.

The "executable image" contains the code and data that make up the executable file. The "relocation information" is a list of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

Microsoft LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, Microsoft LINK creates a temporary disk file to serve as memory. The temporary file is created in the current working directory and is named VM.TMP. When this happens, you will see the following message.

```
VM.TMP has been created.
Do not change diskette in drive, <d:>
```

After this message appears, you must not remove the disk from the given drive (*d*) until the link session ends. If the disk is removed, the operation of Microsoft LINK is unpredictable, and you may see the following message.

```
Unexpected end of file on VM.TMP
```

When this happens you must restart the link session from the beginning.

VM.TMP is a working file only. Microsoft LINK deletes it at the end of the link session.

Warning

Do not give any of your own files the name VM.TMP. If you have a file named VM.TMP on the default drive when Microsoft LINK needs to create a temporary file, LINK deletes the existing VM.TMP file and creates a new one. The contents of the old VM.TMP are lost.

4.3 Linking C Program Files

A number of special considerations should be kept in mind when using Microsoft LINK to process C files. These considerations are discussed below.

4.3.1 The Main Function

When linking C programs, one (and only one) of the object files you submit to Microsoft LINK must have a function named "main". The start-up object module in the standard C library contains a call to the "main" function to begin program execution. If none of the object files you submit contains a "main" function, Microsoft LINK will display an error message informing you that the reference to "main" is unresolved or that the program has no starting address.

4.3.2 Default Libraries and the Library Search Path

Object files created using the Microsoft C Compiler are encoded with the names of the default C libraries for the appropriate memory model. The default C libraries are the standard C library and the floating-point library or libraries selected at compile time. This encoded information enables Microsoft LINK to search for the default library files and link them with your C program.

You do not have to give the names of the default library files when you link. However, you must specify the directory or directories where the library files reside. You can do this by giving directory specifications following the Microsoft LINK "Libraries" prompt, by setting the LIB environment variable, or by combining the two methods.

You can give zero or more directory specifications following the Microsoft LINK "Libraries" prompt. Each directory specification must end with a backslash (\) so Microsoft LINK can recognize the specification as a directory name rather than a library name.

The LIB variable can contain one or more directory specifications. See Section 2.5 of Chapter 2, "Getting Started," for a detailed discussion of environment variables.

To locate library files, Microsoft LINK goes through the following procedure.

1. First, the current working directory is searched.
2. Next, if the library files have not been found, Microsoft LINK searches any directories specified following the Microsoft LINK "Libraries" prompt. The directories are searched in order of their appearance on the line.

3. Finally, if the library files have not been found, Microsoft LINK searches the libraries specified by the LIB environment variable. The directories are searched in order until the given libraries are found.

Note that you can separate the library files and store them in different directories, since Microsoft LINK searches as many of the specified directories as necessary to find the files.

If you want to link with additional libraries, give the library names following the "Libraries" prompt. Microsoft LINK uses the same procedure to search for additional libraries as it does for the default libraries. However, if you give a library name that includes a pathname, Microsoft LINK searches just that pathname for the library; no other directory specifications apply.

4.3.3 Changing the Default Libraries

If you use the /FPa, /FPc87, or /FPc (the default) option when you compile, you are allowed to switch to a different floating-point library at link time. You can do this by giving the name of the library or libraries you want to use following the "Libraries" prompt. See Section 7.7.1 of Chapter 7, "Advanced Topics," for details.

If you do not want to use the standard C library (SLIBC.LIB, MLIBC.LIB, or LLIBC.LIB), you must give the /NOD (for "no default library") option when you link. This option tells Microsoft LINK to ignore the encoded information in the C object files. This option should be used with caution; see the discussion of the /NOD option in Section 4.5.4 for details.

4.3.4 LINK Options to Avoid

Some of the options available with the Microsoft LINK utility are not suitable for use with Microsoft C programs. They include the /HIGH option, the /NOGROUPASSOCIATION option, and the /DSALLOCATE option. Overlays are permitted with C programs, but the /OVERLAYINTERRUPT option (to change the default interrupt number) should not be used.

These options are documented in this chapter along with the other LINK options because you may need them if you use Microsoft LINK to link files written in other languages. The discussion of each option that is not suitable for C programs includes a warning note to that effect.

Using the /DOSSEG option with C programs is not prohibited, but it is never necessary. The segment order specified by the /DOSSEG option is the default segment order for C programs, so the option has no effect on C programs.

4.4 Running the Linker

Microsoft LINK requires two types of input: a command to start Microsoft LINK and responses to command prompts. Start Microsoft LINK by typing

```
LINK
```

at the MS-DOS command level. Microsoft LINK prompts you for the input it needs by displaying the following four lines, one at a time. Microsoft LINK waits for you to respond to each prompt before printing the next one.

```
Object Modules [.OBJ]:
Run File [.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

The responses you can make to each prompt are explained below.

Once you understand the Microsoft LINK prompts and operations, you can use the two alternate methods of running Microsoft LINK. These alternate methods are described in detail below. The command line method lets you type all commands, options, and filenames on the line used to start Microsoft LINK. With the response file method, you create a file that contains all the necessary commands, and then tell Microsoft LINK where to find that file.

Both of the alternate methods require that you understand how Microsoft LINK works and what your responses to its prompts mean. It is recommended that you allow Microsoft LINK to prompt you for responses until you are comfortable with its commands and operations.

You can also invoke LINK through the CL command. See Section C.3 of Appendix C, "The CL Command," for a discussion of linking with the CL command.

4.4.1 Filename Conventions

You can use either uppercase, lowercase, or a combination of both for the filenames you give in response to the prompts. For example, the following three filenames are considered equivalent.

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

Microsoft LINK uses the default file extensions ".OBJ", ".EXE", ".MAP", and ".LIB" when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells Microsoft LINK that the given file has *no* extension, while typing just "ABC" tells Microsoft LINK to use the default extension for that prompt.

4.4.2 Object Modules Prompt

At the "Object Modules" prompt, list the names of the object files you want to link. For C programs, one (and only one) of the object files must contain a "main" function to serve as the entry point for the program. You must respond to this prompt. There is no default.

Microsoft LINK automatically supplies the ".OBJ" extension when you give a filename without an extension. If your object file has a different extension, you must give the full name, with the extension, for the file to be found.

Pathnames are allowed with the object filenames. This means that you can give Microsoft LINK the pathname of an object file in another directory or on another disk. If Microsoft LINK cannot find a given object file, it displays a message and waits for you to change disks.

Each object filename must be separated from the next by blank spaces or a plus sign (+). If a plus sign is the last character typed on the line, the "Object Modules" prompt reappears on the next line, allowing you to give more object files.

4.4.3 Run File Prompt

The "Run File" prompt lets you supply a name for the executable program file. You can give any filename you like; however, it is recommended that you use an ".EXE" or ".COM" extension, since MS-DOS will only execute files having those extensions.

You are allowed to skip this prompt by typing a carriage return without giving a name. By default Microsoft LINK gives the executable file the name of the first ".OBJ" file listed at the previous prompt, with an ".EXE" extension replacing the ".OBJ" extension of the object file.

4.4.4 List File Prompt

At the "List File" prompt you can tell Microsoft LINK to create a listing file. A listing file contains the names of all segments in order of their appearance in the load module. By adding the /MAP option (discussed in Section 4.5.2.1) you can also list all external (public) symbols and their addresses.

If you give a filename without an extension, Microsoft LINK provides the ".MAP" extension. The ".MAP" extension is not required, so you can give another extension if you like. Microsoft LINK creates the listing file in the current working directory unless you give a different pathname.

You can skip this prompt by hitting a carriage return without giving a name. The default response is the special filename NUL.MAP, which tells Microsoft LINK *not* to create a listing file.

4.4.5 Libraries Prompt

Following the "Libraries" prompt you can give zero or more entries, separated by blank spaces or a plus sign (+). If the plus sign is the last character typed, the "Libraries" prompt reappears on the next line, allowing you to type in additional entries. Each entry can be either a directory specification or a library name. Directory specifications must end with a backslash (\) so that Microsoft LINK can distinguish the directory names from the library names.

When you give a directory specification or specifications, Microsoft LINK uses the specifications to search for the default libraries and for any other libraries without a pathname on the same line. To locate the default libraries, Microsoft LINK searches in the following order.

1. In the current working directory
2. In the directories listed following the "Libraries" prompt (in the same order in which they are listed)
3. In the libraries specified by the LIB environment variable

When you give a library name, Microsoft LINK searches for the given library and links it with your program. If the library name includes a directory specification, Microsoft LINK searches only that directory for the library. If just a library name is given (no directory specification), Microsoft LINK uses the search path described above to locate the given library file.

You can give any combination of directory specifications and library names. Notice that you are not required to give any entries; in this case your program will be linked only with the default libraries, and Microsoft LINK will search for the default libraries in the current working directory and in the directories specified by the LIB variable.

Microsoft LINK automatically supplies the ".LIB" extension if you omit it from a library filename. If you want to link a library file with a different extension, be sure to specify the extension.

Microsoft LINK searches all libraries in order of their appearance on the line and searches only until the first definition of a symbol is found. The default libraries are searched after libraries given on the command line are searched. The default floating-point library (or libraries) is searched before the standard C library.

If you do not want to link with the default floating-point library, you can give the name of a different floating-point library instead, provided that you compiled with a "calls" option (/FPc, /FPc87, or /FPa). See Section 3.7 of Chapter 3, "Compiling," for a discussion of floating-point options. If you do not want to link with the standard C library, you must use the /NOD option, discussed in Section 4.5.4.

4.4.6 Separating Entries

Use the plus sign (+) or one or more space characters to separate filename entries in a list of object files or libraries.

4.4.7 Extending Lines

To extend a line, type the plus sign (+) as the last character of a line to be continued. (This is valid only for the "Object Files" and "Libraries" prompts.) The prompt will reappear on the next line, and you can add more entries. Do not type the plus sign in the middle of a filename entry; the plus sign may only be used after complete filenames.

Example

```
Object Modules [.OBJ]: FUN TEXT TABLE CARE+<RETURN>
Object Modules [.OBJ]: YOYO+FLIPFLOP+JUNQUE+<RETURN>
Object Modules [.OBJ]: CORSAIR<RETURN>
```

4.4.8 Selecting Default Responses

To select the default response to the current prompt, type a carriage return without giving a filename. The next prompt will appear.

To select default responses to the current prompt and all remaining prompts, use a single semicolon (;) followed immediately by a carriage return. Once the semicolon has been entered, you cannot respond to any of the remaining prompts for that link session. Use this option to save time when the default responses are acceptable. Note, however, that the semicolon character is not allowed with the first prompt, "Object Modules [.OBJ]", because there is no default response for that prompt.

Defaults for the other linker prompts are as follows.

1. The default for the "Run File" prompt is the name of the first object file submitted for the previous prompt, with the ".EXE" extension replacing the ".OBJ" extension.
2. The default for the "List File" prompt is the special filename NUL.MAP, which tells Microsoft LINK *not* to create a listing file.

3. The default for the "Libraries" prompt for C programs is the floating-point library and the standard C library for the appropriate memory model (small model is the default).

4.4.9 Terminating the Link Session

To terminate the link session at any time, use CONTROL-C. If you type an erroneous response, you may have to type CONTROL-C to exit Microsoft LINK before you can restart the program.

4.4.10 Using the Command Line

To invoke the linker on the command line, give your responses to the command prompts discussed in the last section on a single line, following the LINK command. The responses to the prompts must be separated by commas, as shown below.

LINK *object-list*[, *executable-name*] [, *map-name*] [, *library-list*] [*/option* . . .] [;]

Here *object-list* is a list of object modules, separated by plus signs or blanks. The *executable-name* is the name of the file to receive the executable output. The *map-name* names the file to receive the map listing. The *library-list* consists of libraries and directories to be searched, separated by plus signs or blanks.

The */option* argument or arguments shown at the end of the line are optional. You do not have to give any options when you run the linker. When you do specify options, you can put them after any of the responses in the line. The options available with Microsoft LINK are described in Section 4.5, "Controlling the Linker."

You can select the default response for any prompt by omitting the filename or list before the comma.

You can also select default responses by using the semicolon. The semicolon tells Microsoft LINK to use the default responses for all remaining prompts.

Example

```
LINK FUN+TEXT+TABLE+CARE,  ,FUNLIST,COBLIB.LIB
```

Microsoft LINK loads and links the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ, searching for unresolved references in the library file COBLIB.LIB. By default, the executable file produced is named FUN.EXE. A list file named FUNLIST.MAP is also produced.

4.4.11 Using a Response File

To operate the linker with a response file, you must set up the response file, and then type

`LINK @`*filename*

Here *filename* gives the name of the response file, possibly preceded by a directory specification. You can name the response file anything you like. Microsoft LINK does not impose any naming restrictions for the response file.

A response file contains responses to the Microsoft LINK prompts. Options may be appended to any of the responses or given on a separate line or lines. The responses must be in the same order as the Microsoft LINK prompts discussed above. Each new response must start a new line; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line.

Options and command characters are used in the response file in the same way they are used for responses typed at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, Microsoft LINK uses the default responses for the executable file and for the remaining prompts.

When you give the Microsoft LINK command with a response file, each Microsoft LINK prompt is displayed on your screen with the corresponding response from your response file. If the response file does not contain answers for all the prompts (in the form of filenames, the semicolon command character, or carriage returns), Microsoft LINK displays the missing prompts and waits for you to enter responses. When you type an acceptable response, Microsoft LINK continues the link session.

Example

```
FUN TEXT TABLE CARE
/PAUSE /MAP
FUNLIST
COBLIB.LIB
```

This response file tells Microsoft LINK to load the four object modules FUN, TEXT, TABLE, and CARE. Two output files are produced, named FUN.EXE and FUNLIST.MAP. The /PAUSE option causes Microsoft LINK to pause before producing the executable file (FUN.EXE). This permits you to swap disks if necessary. See the discussion of the /PAUSE and /MAP options in Section 4.5, "Controlling the Linker," for more on these options.

4.5 Controlling the Linker

This section explains how to use linker options to specify and control the tasks performed by the linker. All options begin with the linker option character, the forward slash (/). Options may be placed at the end of any Microsoft LINK response.

When more than one option is given, the options can be grouped at the end of a single response or distributed among several responses. Every option begins with the slash character, even if other options precede it on the same line.

When you use the command line method to invoke Microsoft LINK, options can appear at the end of the line or after individual responses on the line, but before the comma separating each response from the next item. In a response file, options can go after individual responses or by themselves on one of the prompt lines.

The options are named according to their function, with the result that some names are quite long. You can abbreviate the options to save on space and typing. Be sure that your abbreviation is unique so that the linker can determine which option you want. For example, several options begin with the letters "NO" so abbreviations for those options must be longer than "NO" to be unique.

Abbreviations must be sequential from the first letter of the option through the last letter typed. No gaps or transpositions are allowed. To illustrate this rule, the following list shows legal and illegal abbreviations for the /NOIGNORECASE option.

Legal	Illegal
/NOI	/NO
/NOIG	/NIG

Some linker options take numerical arguments. A numerical argument can be any of the following.

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with a zero. For example, the number "10" is a decimal number, but the number "010" is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with "0x". For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

4.5.1 Pausing During Linking

Option

/PAUSE

Unless you instruct it otherwise, Microsoft LINK performs the linking session from beginning to end without stopping. The /PAUSE option tells Microsoft LINK to pause in the link session before writing the executable file to disk. This option allows you to swap disks before Microsoft LINK outputs the executable (".EXE") file.

If the /PAUSE option is given, Microsoft LINK displays the following message before creating the run file.

```
About to generate .EXE file
Change diskette in drive letter: and press <ENTER>
```

The *letter* corresponds to the current drive. Microsoft LINK resumes processing when you press either the ENTER or the RETURN key.

Note

Do not remove the disk that will receive the list file or the disk used for the VM.TMP file, if one has been created.

4.5.2 Producing a Listing File

You can tell Microsoft LINK to produce a listing file by responding to the "List File" prompt. A listing file contains a list of segments in order of their appearance within the load module. The list might look like the following.

```
Start    Stop     Length   Name    Class
00000H   0172CH   0172DH    _TEXT   CODE
01730H   01E19H   006EAH   _DATA   DATA
```

The information in the "Start" and "Stop" columns shows the 20-bit address (in hexadecimal) of each segment relative to the beginning of the load module. The load module begins at location zero. The "Length" column gives the length of the segment in bytes. The "Name" column gives the name of the segment, and the "Class" column gives information about the segment type. For example, a CODE segment contains program code and a DATA segment contains data.

The starting address and name of each group is listed at the end of the file. The group listing might look like the following.

```
Origin   Group
0000:0   IGROUP
0173:0   DGROUP
```

In this example, IGROUP is the name of the code (instruction) group and DGROUP is the name of the data group.

At the bottom of the listing file, Microsoft LINK gives you the address of the program entry point.

4.5.2.1 Listing Public Symbols

Option

/MAP

You can list all public (global) symbols defined in an object file or files by using the /MAP option. The /MAP option forces a listing file to be created if it appears before the "List File" prompt. By default, the file is given the same basename as the executable file plus the extension ".MAP". You can override the default name by responding to the "List File" prompt. However, if the /MAP option appears *after* the "List File" prompt, the option takes effect only if you explicitly created a listing file (by giving a name at the "List File" prompt).

Without the /MAP option the listing file gives only the starting and ending addresses, length, and name of each segment. The /MAP option appends two lists of global symbols to the listing file. The first list is alphabetical by symbol name and the second is sorted by symbol address.

Example

```
Start    Stop     Length   Name    Class
00000H   0172CH   0172DH   _TEXT   CODE
01730H   01E19H   006EAH   _DATA   DATA
   .
   .
   .

Address      Publics by Name
0000:01dB    chkstk
0173:0035    fac
0000:1567    _brk
0000:1696    _chmod
0000_131C    _clearerr
   .
   .
   .

Address      Publics by Value
0000:0035    __chkln
0000:01D2    __fptrap
0000:01DB    __chkstk
0000:023F    _main
0000:025A    __exit
   .
   .
   .
```

The address of the external symbols are in "frame:offset" format, showing the location of the symbol relative to zero (the beginning of the load module).

When you examine a map file, you will notice that the names of globally visible functions and variables begin with an underscore. The Microsoft C compiler automatically prefixes an underscore to all global names to preserve compatibility with XENIX C compilers. If you write assembly language routines to interface with your C program, this naming convention is important; see Section 8.1.6 of Chapter 8, "Interfaces with Other Languages."

In the listing file, you may also see names that begin with more than one underscore. Identifiers with more than one leading underscore are reserved for internal use by the compiler. You should not attempt to use these identifiers in your program. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically

adds another leading underscore, these names end up with two leading underscores, possibly causing conflicts with the names reserved by the compiler.

4.5.2.2 Displaying Line Numbers

Option

/LINENUMBERS

You can include the line numbers and associated addresses of your source program in the linker listing by using the /LINENUMBERS option. Ordinarily the linker listing does not contain line numbers.

To produce a linker listing with line numbers, you must give Microsoft LINK an object file (or files) with line number information. With the C compiler you can use the /Zd switch to produce line numbers in the object file. If you give Microsoft LINK an object file without line number information, the /LINENUMBERS option has no effect.

The /LINENUMBERS option forces a listing file to be created if it appears before the "List File" prompt. By default, the file is given the same basename as the executable file plus the extension ".MAP". You can override the default name by responding to the "List File" prompt. However, if the /LINENUMBERS option appears *after* the "List File" prompt, the option takes effect only if you explicitly created a listing file (by giving a name at the "List File" prompt).

4.5.3 Ignoring Case

Option

/NOIGNORECASE

By default, Microsoft LINK treats uppercase and lowercase letters as equivalents. Thus, "ABC", "abc", and "Abc" are considered to be the same name. When you use the /NOIGNORECASE option (usually abbreviated /NOI), the linker distinguishes between uppercase and lowercase letters and considers "ABC", "abc", and "Abc" three separate names.

The C language is case-sensitive: two names are identical only if they have exactly the same letters in the same case. If your C programs rely on this behavior, you should use the /NOIGNORECASE option. Keep in mind, however, that some programs, such as assemblers and other language compilers, may not make case distinctions. If you want to link such programs with your C programs, it is best to give each identifier a unique spelling to avoid conflicts.

4.5.4 Ignoring Default Libraries

Option

/NODEFAULTLIBRARYSEARCH

The /NODEFAULTLIBRARYSEARCH option (usually abbreviated to /NOD) tells Microsoft LINK *not* to search the default library, if there is one, to resolve external references. With C files this has the effect of telling Microsoft LINK to ignore the information in the object files that gives the names of the standard C library and selected floating-point library.

Most C programs will not work correctly without the standard C library, so if you use the /NOD option you should explicitly specify the name of the standard library, as well as any floating-point libraries needed by the program. If you do not use the standard library, you must provide your own start-up routine, or extract the start-up routine from the standard library and link it with your program. (See Section 2.4.3, "Library Files," in Chapter 2, "Getting Started," for a discussion of the start-up routine.)

When using the /NOD option with C programs, always use the following order to specify libraries.

1. Any libraries other than the standard C library or floating-point libraries
2. The floating-point library or libraries
3. The standard C library

4.5.5 Controlling Stack Size

Option

/STACK: *number*

The /STACK option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal, or hexadecimal) up to 65,536 (decimal). It represents the size, in bytes, of the stack.

All compilers and assemblers should provide information in the object modules that tells the linker how to set up the stack. For C programs, the default stack size is 2K bytes. The default stack size is set by the start-up routine (CRT0.OBJ) in the standard C library.

If your program has a large amount of local data or is heavily recursive, you may get a stack overflow message. In this case you need to increase the size of the stack. On the other hand, if your program uses very little local data, you may achieve some space savings by decreasing the stack size.

If Microsoft LINK cannot find the stack information it needs, it displays the following error message. Since the start-up file provides stack information, this message usually means that the start-up file is not being linked with your program.

```
WARNING: NO STACK SEGMENT
```

Note

The EXEMOD utility (described in Section 7.9.2 of Chapter 7, "Advanced Topics") can also be used to change the default stack size for C program files.

4.5.6 Allocating Paragraph Space

Option

/CPARMAXALLOC : *number*

This option allows you to change the default value of the "cparMaxALLOC" field, which controls the maximum number of paragraphs in memory allocated for your program. A paragraph is defined as the smallest memory unit addressed by a segment register, or 16 bytes.

The maximum number of paragraphs allocated for a program is determined by the value of the "cparMaxALLOC" field at offset 0x0c in the EXE header. See your *Microsoft MS-DOS Programmer's Reference Manual* for details.

By default, the "cparMaxALLOC" field is set to 65,535 (decimal), or 64K minus 1. The /CPARMAXALLOC option lets you set the value to any number between 1 and 65,535 by giving the desired *number* (decimal, octal, or hexadecimal) with the option. This is useful when you know that the efficiency of your program will not be increased by allocating all available memory, or when you want to execute another program from within your program and you need to reserve space for the executed program.

If the value you specify is less than the computed value of "cparMinAlloc" (at offset 0A hexadecimal), the linker uses the value of "cparMinAlloc" instead.

4.5.7 Controlling Segments

Option

/SEGMENTS : *number*

The /SEGMENTS option controls the number of segments the linker allows a program to have. The default is 128. The *number* can be set to any value (decimal, octal, or hexadecimal) in the range 1 to 1,024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), the linker avoids having to allocate a large amount of storage space for all programs.

When you set the segment limit higher than 128, the linker correspondingly allocates more space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting the segment *number* to reflect the actual number of segments in the program.

4.5.8 Using Overlays

You can direct Microsoft LINK to create an overlaid version of your program. This means that parts of your program will only be loaded if and when they are needed, and will share the same space in memory. Your program will be smaller as a result, but will usually run more slowly because of the time needed to read and reread the code into memory.

Provided your modules obey the restrictions described below, all you have to do is specify the overlay structure to the linker. Loading of the overlays is automatic. You specify overlays in the list of modules that you submit to the linker by enclosing them in parentheses. Each parenthetical list represents one overlay. For example, in the following response to the "Object Modules" prompt,

```
Object Modules [.OBJ] a + (b+c) + (e+f) + g + (i)
```

the elements "(b+c)", "(e+f)", and "(i)" are overlays. The remaining modules, and any drawn from the run-time libraries, make up the resident part of your program, or root. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can occur only once in a program.

The linker will replace calls from the root to an overlay and calls from an overlay to another overlay with an interrupt (followed by the module identifier and offset.) The interrupt number is 3F hexadecimal.

4.5.8.1 Restrictions

The name for the overlays is appended to the ".EXE" file, and the name of this file is encoded into the program so the Overlay Manager can access it. If, when the program is initiated, the overlay manager cannot find the ".EXE" file (perhaps you have renamed it or it is not in a directory specified by the PATH environment variable), then the overlay manager will prompt you for a new name.

You can only overlay modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. With C programs, long calls are the default only in medium and large model programs. See Section 3.13 of Chapter 3, "Compiling," for details on the standard memory models.

You cannot use long jumps (via the *longjmp* library function) or indirect calls (through a function pointer) to pass control to an overlay. When a function is called through a pointer, the called function must be in the same overlay or in the root.

4.5.8.2 Overlay Manager Prompts

Suppose that B: is the default drive. In the following example,

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

the response EMPLOYEE\DATA\ causes the overlay manager to look for EMPLOYEE\DATA\PAYROLL.EXE on B.

Now, suppose that it becomes necessary to change the disk in Drive B. If the overlay manager needs to swap overlays, it will find that PAYROLL.EXE is no longer on B, and will give the following message.

```
Please put diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After the overlay has been read from the disk, the overlay manager will give the following message.

```
Please restore the original diskette.
Strike any key when ready.
```

4.5.9 Setting the Overlay Interrupt

Option

/OVERLAYINTERRUPT:*number*

By default, the interrupt number used for passing control to overlays is 3F hexadecimal. The overlay interrupt option allows the user to select a different interrupt number. The *number* can be a decimal number from 0 to 255, an octal number from 0 to 0377, or a hexadecimal number from 0 to 0xFF. Numbers that conflict with MS-DOS interrupts are not prohibited, but their use is not advised.

Warning

Do not use the /OVERLAYINTERRUPT option with C programs.

4.5.10 Ordering Segments

Option

/DOSSEG

The /DOSSEG switch forces segments to be ordered according to the following rules.

1. All segments with a class name ending in CODE come first.
2. All other segments outside of DGROUP come next.
3. DGROUP segments come last, in the following order.
   a. Any segments of class BEGDATA (this class name is reserved for Microsoft use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

See Section 7.13 of Chapter 7, "Advanced Topics," for a discussion of the segment names used by the C compiler.

4.5.11 Controlling Data Loading

Option

/DSALLOCATE

By default, Microsoft LINK loads all data starting at the low end of the data segment. At run time, the DS (data segment) pointer is set to the lowest possible address to allow the entire data segment to be used.

Use the /DSALLOCATE option to tell Microsoft LINK to load all data starting at the high end of the data segment instead. In this case the DS pointer is set at run time to the lowest data segment address that contains program data.

The /DSALLOCATE option is typically used with the /HIGH option, discussed in the next section, to take advantage of unused memory within the data segment. The user can allocate any available memory below the area specifically allocated for DGROUP, using the same DS pointer.

Warning

Do not use the /DSALLOCATE option with C programs.

4.5.12 Controlling Run File Loading

Option

/HIGH

The run file can be placed either as low or as high in memory as possible. Use of the /HIGH option causes Microsoft LINK to place the run file as high as possible in memory. Without the /HIGH option, Microsoft LINK places the run file as low as possible.

Warning

Do not use the /HIGH option with C programs.

4.5.13 Preserving Compatibility

Option

/NOGROUPASSOCIATION

The /NOGROUPASSOCIATION option causes the linker to process a certain class of fix-ups in a manner that is compatible with previous versions of the linker (Versions 2.02 and earlier). It is provided primarily for compatibility with previous versions of other Microsoft language compilers.

Warning

Do not use the /NOGROUPASSOCIATION option with C programs.

Chapter 5
Running Your C Program

5.1 Running a Program 111
5.2 Passing Data to a Program 111
5.3 Expanding Wild Card Arguments 114
5.4 Suppressing Command Line Processing 115
5.5 Suppressing Null Pointer Checks 116

5.1 Running a Program

Once you have created an executable file, you can run your program by giving the name of the file without the extension. However, the file must have a ".EXE" or ".COM" extension; otherwise MS-DOS will not execute it.

MS-DOS uses the PATH environment variable to find executable files. You can execute a program from any directory, as long as the executable program file is either in your current working directory or in one of the directories on the PATH.

The *spawn, exec,* and *system* routines provided in the Microsoft C Run-Time Library let you execute other programs and MS-DOS commands from within a program. See the *Microsoft C Run-Time Library Reference* for a description of these routines.

Example

```
MYPROG
```

This example runs the executable file named MYPROG.EXE (or MYPROG.COM).

5.2 Passing Data to a Program

You can access data on the command line or in the environment table at execution time by declaring arguments to the "main" function. Command line data are any data that appear on the same line as the program name when you execute the program. The environment table contains all environment settings in effect at execution time (see Section 2.5, "Setting Up the Environment," in Chapter 2, "Getting Started," for a discussion of environment variables). Since the "main" function is where program execution begins, "main" is the function that takes charge of data pased at execution time.

To pass data to your program on the command line, give one or more arguments after the program name when you execute it. Each argument must be separated from the arguments around it by one or more spaces or tab characters, and may be enclosed in quotation marks (""). If you want to give a single argument that includes spaces or tab characters, you must enclose the argument in quotation marks. For example, the command line

```
TEST 42 "de f" 16
```

executes the program named TEST.EXE (or TEST.COM) and passes three arguments: "42", "de f", and "16".

Under MS-DOS, each command line argument, regardless of its data type, is stored as a null-terminated string in an array of strings. The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

To set up your program to receive the command-line data, declare arguments to the "main" function, as shown below. By declaring these variables as arguments to "main", you make them available as local variables in the "main" function.

```
main (argc, argv, envp)

int argc;
char *argv[];
char *envp[];
```

You do not have to declare all three arguments. However, you must declare the arguments in the order shown above. Thus, if you want to use the "envp" arguments, you must declare "argc" and "argv", even if you do not use them.

The command line is passed to the program as the "argv" array of strings. The number of arguments appearing on the command line is passed as the integer variable "argc".

The first argument of any command line is the name of the program to be executed. Thus, the program name is the first string stored in "argv", at "argv[0]". Since a program name must be given in all cases, the integer value of "argc" is always at least one. The first argument given after the program name is stored at "argv[1]", the second is stored at "argv[2]", and so on through the end of the arguments. The total number of arguments, including the program name, is stored in "argc".

Note

Under versions of MS-DOS earlier than 3.0, the program name normally stored in "argv[0]" is not available. References to "argv[0]" yield the empty string. Under MS-DOS Version 3.0 and later, references to "argv[0]" give the program name.

The third argument passed to the "main" function, "envp", is a pointer to the environment table. You can access the value of environment settings through this pointer. However, the *putenv* and *getenv* routines from the C run-time library accomplish the same task, and are easier and safer to use. Use of the *putenv* routine may change the location of the environment table in memory, depending on memory requirements. Thus, the value given to "envp" at the beginning of the program execution may not be valid throughout the program's execution. The *putenv* and *getenv* routines, on the other hand, access the environment table properly, even when its location changes. These routines use the global variable *environ* (described in the *Microsoft C Run-Time Library Reference*), which always points to the correct table location.

Example

```
MYPROG ABC "abc e" 3 8
```

This command line executes the program named MYPROG and passes the four command line arguments to the "main" function. The arguments are stored as null-terminated strings, and the number of arguments is stored in "argc". To access the last argument, for example, you would use an expression like the following.

```
argv[argc - 1]
```

Since the value of "argc" is 5 (counting the program name as an argument), this expression is equivalent to "argv[4]", or the fifth string of the array.

5.3 Expanding Wild Card Arguments

You can use the MS-DOS wild card characters, the question mark (?) and the asterisk (*), to specify filename and pathname arguments on the command line. To prepare for using wild cards, you must link with the special routine SSETARGV.OBJ (MSETARGV.OBJ or LSETARGV.OBJ for medium or large model programs). These object files are included with your compiler software. If you don't link with the appropriate object file, wild characters are treated literally.

The SSETARGV.OBJ expands the wild card characters in the same manner as MS-DOS. (See your MS-DOS documentation if you are unfamiliar with these characters.) Enclosing an argument in quotation marks (" ") suppresses the wild card expansion. Within quoted arguments, you can represent quotation marks literally within an argument by preceding the double quote character with a backslash (\).

If no matches are found for the wild card argument, the argument is passed literally. For example, if the argument B:*.C is given, but no files with the extension ".C" are found in the root directory of Drive B, the argument is passed as the string "B:*.C".

If you frequently use wild card expansion, you may want to place the wild card routines (SSETARGV.OBJ, MSETARGV.OBJ, and LSETARGV.OBJ) in the appropriate standard C libraries (SLIBC.LIB, MLIBC.LIB, and LLIBC.LIB). That way the routine will be linked with your program automatically. To do this, use the LIB utility (described in Chapter 6, "Managing Libraries") to extract the module named *stdargv* from the library (the module name is the same in all three libraries) and insert SSETARGV, MSETARGV, or LSETARGV, as appropriate. When you replace *stdargv* with the appropriate SETARGV routine, wild card expansions will always be performed on command line arguments.

Example

```
LINK BETA SSETARGV;
BETA *.INC "WHY?" \"HELLO\"
```

In this example, SSETARGV.OBJ is linked with BETA.OBJ, producing the executable file BETA.EXE. When BETA.EXE is executed, the wild card character "*" is expanded, causing all filenames with the extension ".INC" in the current working directory to be passed as arguments to the

BETA program. The second command line argument, WHY?, is enclosed in quotation marks (" "), so expansion of the wild card character "?" is suppressed and the argument WHY? is passed literally. In the third argument, the backslashes cause the quotations to be represented literally so the argument "HELLO" is passed.

5.4 Suppressing Command Line Processing

If your program does not take command line arguments, you can achieve a small space savings by suppressing use of the library routine that performs command line processing. This routine is called *_setargv*. To suppress its use, define a routine that does nothing in the same file that contains the "main" function, and name it *_setargv*. The call to *_setargv* will be satisfied by your definition of *_setargv*, and the library version will not be loaded.

Similarly, if you never access the environment table through the "envp" argument, you can provide your own empty routine to be used in place of *_setenvp*, the environment processing routine. In the same file that contains the "main" function, define a routine that does nothing and name it *_setenvp*.

If your program makes calls to the *spawn* or *exec* routines in the C run-time library, you may not want to suppress the environment processing routine. This routine is used to pass an environment from the parent process to the child process.

Example

```
_setargv()
{
}

_setenvp()
{
}
```

The above example shows how to define the *_setargv* and *_setenvp* functions to suppress command line and environment processing. It is recommended that you place these definitions in the file containing the "main" function.

5.5 Suppressing Null Pointer Checks

When you execute your C program, a special error-checking routine is automatically invoked to determine whether the contents of the NULL segment have changed and to display the following error message if they have.

```
Null pointer assignment
```

The NULL segment is a special location in low memory that is not normally used. If the contents of the NULL segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Notice that your program can contain null pointers without generating this message; the message appears only when you write to a memory location through the null pointer.

This error does not cause your program to terminate; the error is detected and the error message is printed following the normal termination of the program.

Important

The "null pointer assignment" message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

The library routine that performs the null pointer check is named *_nullcheck*. You can suppress the null pointer check for a particular program by defining your own routine named *_nullcheck* that does nothing. The call to *_nullcheck* will be satisfied by your definition of *_nullcheck*, and the library version will not be loaded. It is recommended that you place the *_nullcheck* definition in the file containing the "main" function.

To suppress the null pointer check for all programs, you can replace the corresponding error-checking routine in the standard C library. The routine is stored in a module called *chksum* in all three libraries (SLIBC.LIB, MLIBC.LIB, and LLIBC.LIB). Do not remove the routine entirely or there will be an unresolved reference in your program. Instead, use the LIB

utility (described in Chapter 6, "Managing Libraries") to replace the *chksum* module with a module containing the empty definition of *_nullcheck*. This replacement will satisfy the call to *_nullcheck* and null pointer checking will not be performed.

Chapter 6
Managing Libraries

6.1 Introduction 121
6.2 Overview of LIB Operation 122
6.3 Running LIB 123
6.3.1 Library Name Prompt 124
6.3.2 Operations Prompt 125
6.3.3 List File Prompt 126
6.3.4 Output Library Prompt 127
6.3.5 Using the Command Line 127
6.3.6 Using a Response File 128
6.3.7 Extending Lines 129
6.3.8 Terminating the Library Session 129
6.3.9 Selecting Default Responses to Prompts 130
6.4 Library Tasks 130
6.4.1 Creating a Library File 130
6.4.2 Modifying a Library File 131
6.4.3 Adding Library Modules 131
6.4.4 Deleting Library Modules 132
6.4.5 Replacing Library Modules 132
6.4.6 Extracting Library Modules 132

6.4.7 Combining Libraries 133
6.4.8 Creating a Cross-Reference Listing 133
6.4.9 Performing Consistency Checks 134
6.4.10 Setting the Library Page Size 134

6.1 Introduction

The Microsoft LIB Library Manager is a utility designed to help you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. Any program can call a run-time routine, exactly as if the function were included in the program. The program is linked with the run-time library file and the call to the run-time routine is resolved by finding the routine in the library file.

Run-time libraries are created by combining separately compiled object files into one library file. Library files are usually identified by their ".LIB" extension, although other extensions are allowed.

In addition to accepting MS-DOS object files and library files, Microsoft LIB accepts 286 XENIX archives and Intel-style libraries. You can add the contents of a 286 XENIX archive or an Intel-style library to an MS-DOS library by using the append operator (+).

Once an object file is incorporated into a library, it becomes an object "module." LIB makes a distinction between object files and object modules: an object "file" exists as an independent file while an object "module" is part of a larger library file. An object file can have a full pathname, including a drive designation, directory pathname, and filename extension (usually ".OBJ"). Object modules have just a name. For example, "B:\RUN\SORT.OBJ" is an object filename, while "SORT" is the corresponding object module name.

Using LIB, you can create a new library file, add object files to an existing library, delete library modules, replace library modules, and create object files from library modules. LIB also lets you combine the contents of two libraries into one library file.

The command syntax is straightforward, and LIB prompts you for responses. Once you have learned how LIB works and what its prompts mean, you can use one of the alternate methods of invoking LIB, described in the sections entitled "Using the Command Line" and "Using a Response File" later in this chapter. The alternative methods let you give LIB commands without waiting for the LIB prompts.

6.2 Overview of LIB Operation

You can perform a number of library management functions with Microsoft LIB. LIB can

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Append an object file as a module of a library, or append the contents of a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules

For each library session, LIB first reads and interprets the user's commands. It determines whether a new library is being created or an existing library is being examined or modified.

Deletion and extraction commands (if any) are the first commands processed. LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

Next, LIB processes any addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or extraction commands, a new library file is created in the addition stage by copying the original library file.)

As LIB carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The library index is used by the linker to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. LIB produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Thus, when you terminate LIB for any reason, you do not lose your original file. It also means that when you run LIB, enough space must be available on your disk for both the original library file and the copy.

When you modify a library file, LIB gives you the option of specifying a different name for the file containing the modifications. If you use this option, the modified library is stored under the name you give, and the original, unmodified version is preserved under its own name. If you choose not to give a new name, LIB gives the modified file the original library name, but keeps a backup copy of the original library file. This copy has the extension ".BAK" instead of ".LIB".

Example

```
LIB LANG-HEAP+HEAP;
```

This command first deletes the library module HEAP from the library file LANG.LIB, then adds the file HEAP.OBJ as the last module in the library. The same command could be given as

```
LIB LANG+HEAP-HEAP;
```

The effect is the same because delete operations are always carried out before append operations, regardless of the order of the operations in the command line. This order of execution prevents confusion in LIB when a new version of a module replaces an old version in the library file.

After the library is modified, the modified file is written back to LANG.LIB. A copy of the original LANG.LIB is stored in LANG.BAK.

6.3 Running LIB

LIB requires two types of input: a command to start LIB and responses to command prompts. Start LIB at the MS-DOS command level by typing

```
LIB
```

LIB prompts you for the input it needs by displaying the following four

messages, one at a time. LIB waits for you to respond to each prompt, then prints the next prompt.

```
Library name:
Operations:
List file:
Output library:
```

The responses you can make to each prompt are explained in the following four sections.

Once you understand the LIB prompts and operations, you may want to use one of the two alternate methods of running LIB. The command line method lets you type all commands, options, and filenames on the line used to start LIB. With the response file method, you create a file that contains all the necessary commands, then tell LIB where to find that file.

Both of the alternate methods require that you understand how LIB works and what your responses to its prompts mean. For this reason it is recommended that you allow LIB to prompt you for responses until you are comfortable with its commands and operations.

6.3.1 Library Name Prompt

At the "Library name" prompt, give the name of the library file you want. Usually library files are named with the ".LIB" extension. You can omit the ".LIB" extension when you give the library filename since LIB assumes that the filename extension is ".LIB".

If your library file does not have the ".LIB" extension, be sure to include the extension when you give the library filename. Otherwise, LIB cannot find the file.

Pathnames are allowed with the library filename, so you can give LIB the pathname of a library file in another directory or on another disk.

Because LIB manages only one library file at a time, only one filename is allowed in response to this prompt. There is no default response, so LIB produces an error message if you do not give a filename.

If you give the name of a library file that does not exist, LIB displays the prompt:

```
Library file does not exist.  Create?
```

Type "y" to create the library file. If you answer "n", LIB returns control to the MS-DOS level.

If you type a library filename and follow it immediately with a semicolon (;), LIB performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. LIB prints a message only if it finds an invalid object module; no message appears if all modules are intact.

You can also set the library page size following this prompt. For details, see Section 6.4.10 "Setting the Library Page Size."

6.3.2 Operations Prompt

Following the "Operations" prompt, you can type one of the command symbols for manipulating modules (+, -, -+, *, -*) followed immediately by the module name or the object filename. You can specify more than one operation following this prompt, in any order. The default for the "Operations" prompt is no changes.

When you have a large number of modules or files to manipulate (more than can be typed on one line), type an ampersand (&) as the last symbol on the line, immediately before the carriage return. The ampersand must follow a filename; you cannot give an operator as the last character on a line to be continued. The ampersand causes LIB to repeat the "Operations" prompt, allowing you to specify more operations and names.

The following list describes command symbols and their meanings and uses.

Command Symbol	Meaning and Use
+	The plus sign makes an object file the last module in the library file. Give the name of the object file immediately after the plus sign. You can use pathnames for the object file. LIB automatically supplies the ".OBJ" extension, so you can omit the extension from the object filename. You can also use the plus sign to combine two libraries. When you give a library name after the plus sign, a copy of the contents of the given library is added to the library file being modified. You must include the ".LIB"

extension when you give a library filename. Otherwise, LIB uses the default ".OBJ" extension when it looks for the file.

`-` The minus sign deletes a module from the library file. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension.

`-+` Type a minus sign followed by a plus sign to replace a module in the library. Give the name of the module to be replaced after the replacement symbol. Module names have no pathnames and no extensions.

To replace a module, LIB first deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

`*` Type an asterisk followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

`-*` Use the minus sign followed by an asterisk to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

6.3.3 List File Prompt

After the "List file" prompt, you can give a filename for a cross-reference listing file. You can specify a full pathname for the listing file to cause it to be created outside your current working directory. You can give the listing file any name and any extension. LIB does not supply a default extension if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all external (public) symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is a list of the modules in the library. Under each module name is an alphabetical listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special filename NUL., which tells LIB *not* to create a listing file.

6.3.4 Output Library Prompt

After the "Output library" prompt you can give the name of a new library file to create with the specified modifications. The default is the current library filename; the original, unmodified library is saved in a library file with the same name but with a ".BAK" extension replacing the ".LIB" extension. This prompt appears only if you specify modifications to the library following the "Operations" prompt.

6.3.5 Using the Command Line

The command line method of starting LIB has the form

`LIB` *library* `[`*/pagesize*`]` `[;]` *operations*`...` `[,` *listing*`]` `[,` *newlibrary*`]`

The entries following LIB are responses to the LIB command prompts.

The *library* entry, with the optional */pagesize* specification, corresponds to the "Library name" prompt. If you want LIB to perform a consistency check on the library, follow the *library* entry with a semicolon (;).

The *operations* entries are any of the operations allowed following the "Operations" prompt. The *listing* entry, if you include it, tells LIB to create a listing file with the given name. The *newlibrary* entry, if it appears, is the name of the revised library.

If you want to create a cross-reference listing, the name of the listing file must be separated from the last *operations* entry by a comma. If you give a filename in the new library field, the library name must be separated from the listing filename or the last *operations* entry by a comma.

You can use a semicolon after any entry but the first to tell LIB to use the default responses for the remaining entries. The semicolon should be the last character on the command line.

Examples

```
1.  LIB LANG-+HEAP;

2.  LIB C;

3.  LIB LANG,LCROSS.PUB
```

The first command instructs LIB to replace the HEAP module in the library LANG.LIB. LIB first deletes the HEAP module in the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon command symbol at the end of the command line tells LIB to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written back to the original library file instead of creating a new library file.

The second command causes LIB to perform a consistency check of the library file C.LIB. No other action is performed. LIB displays any consistency errors it finds and returns to the operating system level. The third command tells LIB to perform a consistency check of the library file LANG.LIB, then output a cross-reference listing file named LCROSS.PUB.

6.3.6 Using a Response File

The command to start LIB with a response file has the form

LIB @*response-file*

The *response-file* field is the name of a response file. The *response-file* name may be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

Before you use this method you must set up a response file containing answers to the LIB prompts. This method lets you conduct the library session without typing responses at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would for responses typed on the keyboard.

When you run LIB with a response file, the prompts are displayed along with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses.

Example

```
SLIBC
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes LIB to delete the module HEAP from the SLIBC.LIB library file; extract the module FOIBLES and place it in an object file named FOIBLES.OBJ; and then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference file named CROSSLST.

6.3.7 Extending Lines

If you have many operations to perform during a library session, use the ampersand (&) command symbol to extend the operations line. Give the ampersand symbol after an object module or object filename; do not put the ampersand between an operations symbol and a name.

If you use the ampersand with the prompt method of invoking LIB, the ampersand will cause the "Operations" prompt to be repeated, allowing you to type in more operations. With the response file method, you can use the ampersand at the end of a line and then continue typing operations on the next line.

6.3.8 Terminating the Library Session

At any time, you can use CONTROL-C to terminate a library session. If you type an incorrect response, such as a wrong or incorrectly spelled filename or module name, you must enter CONTROL-C to exit LIB. You can then restart the program.

6.3.9 Selecting Default Responses to Prompts

After any entry but the first, use a single semicolon (;) followed immediately by a carriage return to select default responses to the remaining prompts. You can use the semicolon command symbol with the command line and response file methods of invoking LIB, but it is not really necessary, since LIB supplies the default responses wherever you omit responses.

The default response for the "Operations" prompt is no operation. The library file is unchanged.

The default response for the "List file" prompt is the special filename NUL., which tells LIB not to create a listing file.

The default response for the "Output library" file is the current library name. This prompt appears only if you specify at least one operation following the "Operations" prompt.

6.4 Library Tasks

This section summarizes the library management tasks you can perform with LIB.

6.4.1 Creating a Library File

To create a new library file, simply give the name of the library file you want to create following the "Library name" prompt. LIB supplies the ".LIB" extension.

The name of the new library must not be the name of an existing file, or else LIB will assume you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt.

```
Library file does not exist.  Create?
```

Type "yes" to create the file, "no" to terminate the library session.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See the section on setting the page size below for a discussion of this option.

Once you have given the name of the new library file, you can insert object modules into the library by using the add operation (+) following the "Operations" prompt. You can also add the contents of another library if you wish. These options are discussed below in the sections entitled "Adding Library Modules" and "Combining Libraries," respectively.

6.4.2 Modifying a Library File

You can modify an existing library file by giving the name of the library file following the "Library name" prompt. All operations you specify following the "Operations" prompt are performed on that library.

However, LIB lets you keep both the unmodified library file and the newly modified version, if you like. You can do this by giving the name of a new library file following the "Output library" prompt. The modified library file is stored under the new library filename, while the original library file remains unchanged.

If you don't give a filename following the "Output library" prompt, the modified version of the library file replaces the original library file. Even in this case, LIB saves the original, unmodified library file. The unmodified library file has the extension ".BAK" instead of ".LIB". Thus, at the end of the session you have two library files: the modified version with the ".LIB" extension and the original, unmodified version with the ".BAK" extension.

6.4.3 Adding Library Modules

Use the plus sign (+) following the "Operations" prompt to add an object module to a library. Give the name of the object file to be added, without the ".OBJ" extension, immediately after the plus sign.

LIB strips the drive designation and the extension from the object file specification, leaving only the basename. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is "CURSOR".

Object modules are always added to the end of a library file.

6.4.4 Deleting Library Modules

Use the minus sign (-) following the "Operations" prompt to delete an object module from a library. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension; it is simply a name, like "CURSOR".

6.4.5 Replacing Library Modules

Use a minus sign followed by a plus sign (-+) to replace a module in the library. Give the name of the module to be replaced after the replacement symbol (-+). Remember that module names have no pathnames and no extensions.

To replace a module, LIB first deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

6.4.6 Extracting Library Modules

Use an asterisk (*) followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

Use the minus sign followed by an asterisk (-*) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

6.4.7 Combining Libraries

You can add the contents of a library to another library by using the plus sign (+) with a library filename instead of an object filename. Following the "Operations" prompt, give the plus sign (+) followed by the name of the library whose contents you wish to add to the library being modified. When you use this option you must include the ".LIB" extension of the library filename. Otherwise, LIB assumes that the file is an object file and looks for the file with an ".OBJ" extension.

In addition to allowing MS-DOS libraries as input, LIB also accepts 286 XENIX archives and Intel-format libraries. Thus, you can use LIB to convert libraries from either of these formats to the Microsoft format.

LIB adds the modules of the library to the end of the library being modified. Notice that the added library still exists as an independent library. LIB copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name following the "Output library" prompt. If you omit the "Output library" response, LIB saves the combined library under the name of the original library being modified.

6.4.8 Creating a Cross-Reference Listing

Create a cross-reference listing by giving a name for the listing file following the "List file" prompt. If you omit the response to this prompt, LIB uses the special filename NUL., which means that no listing file is created.

You can give the listing file any name and any extension. You can specify a full pathname, including drive designation, for the listing file to cause it to be created outside your current working directory. LIB does not supply a default extension if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols referenced in that module.

6.4.9 Performing Consistency Checks

When you give just a library name followed by a semicolon following the "Library name" prompt, LIB performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. This option is not usually necessary, since LIB automatically checks object files for consistency before adding them to the library.

To produce a cross-reference listing along with a consistency check, use the command line method of invoking LIB. Give the library name followed by a semicolon, then give the name of the listing file. LIB performs the consistency check and then creates the cross-reference listing.

6.4.10 Setting the Library Page Size

The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned so that they always start at a position that is a multiple of *n* bytes from the beginning of the file. The value of *n* is the page size. The default page size is 16 for a new library or the current page size for an existing library.

Because of the indexing technique used by LIB, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of $n/2$ bytes of storage space is wasted (where *n* is the page size). In most cases a small page size is advantageous, and you are advised to use the smallest page size possible.

To set the library page size, add a page size option after the library filename following the "Library name" prompt:

library `/pagesize:`*n*

The value of *n* is the new page size. It must be a power of 2 and must fall between 16 and 32,768.

Chapter 7
Advanced Topics

7.1 Introduction 137
7.2 Enabling Special Keywords 137
7.3 Packing Structure Members 137
7.4 Restricting Length of External Names 138
7.5 Labeling the Object File 139
7.6 Suppressing Default Library Selection 139
7.7 Controlling Floating-Point Operations 140
7.7.1 Changing Libraries at Link Time 141
7.7.2 Using the NO87 Environment Variable 142
7.8 Advanced Optimizing 143
7.8.1 Removing Stack Probes 144
7.8.2 Maximum Optimization 145
7.9 Modifying the Executable File 145
7.9.1 The EXEPACK Utility 146
7.9.2 The EXEMOD Utility 146
7.10 Controlling Binary and Text Modes 147

7.11 Mixed Model Programming 148
7.11.1 Using the Near and Far Keywords 150
7.11.2 Creating Customized Memory Models 151
7.11.2.1 Code Pointers 151
7.11.2.2 Data Pointers 152
7.11.2.3 Setting Up Segments 152
7.11.3 Library Support 153
7.12 Setting the Data Threshold 154
7.13 Naming Modules and Segments 155

7.1 Introduction

The Microsoft C Compiler offers a number of advanced programming options that give you control over the compilation process and the final form of the executable program. This chapter describes the advanced options.

7.2 Enabling Special Keywords

Option

/Ze

The /Ze option tells the compiler to consider the following identifiers keywords when processing a given file.

far
fortran
huge
near
pascal

You must give this option when compiling any program that uses these identifiers as keywords. The **huge** keyword is not yet implemented but is reserved for future use.

7.3 Packing Structure Members

Option

/Zp

When storage is allocated for structures, structure members larger than a **char** are ordinarily stored beginning at an **int** boundary. To conserve space you may want to store your structures more compactly. The /Zp option causes structure data to be "packed" tightly into memory. This option is also useful when you want to read existing packed structures from a data file.

When you give the /Zp option, each structure member (after the first) is stored beginning at the first available byte, without regard to int boundaries.

On most processors, using the /Zp option results in slower program execution because of the time required to unpack structure members when they are accessed. This option also reduces efficiency when a program accesses 16-bit members (with int type) that begin on odd boundaries.

Example

```
MSC /Zp PROG.C;
```

This command causes all structures in the program PROG.C to be stored without extra space for alignment of members on int boundaries.

7.4 Restricting Length of External Names

Option

/Hnumber

The MSC command allows you to restrict the length of external (public) names by using the /H option. The *number* is an integer specifying the maximum number of significant characters in external names.

When you use the /H option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters; the extra characters are simply ignored.

The /H option is typically used to conserve space or to aid in creating portable programs. The Microsoft C Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

7.5 Labeling the Object File

Option

/V"string"

Use the /V (for "version") option to imbed a given text *string* into an object file. The quotation marks surrounding the string may be omitted if the string does not contain whitespace characters.

Object files are machine readable but are not easily read and understood by humans. A typical use of the /V option is labeling an object file with a version number or copyright notice.

Example

```
MSC MAIN.C, /V"Microsoft C Compiler Version 3.0";
```

The above command places the string "Microsoft C Compiler Version 3.0" in the object file MAIN.OBJ.

7.6 Suppressing Default Library Selection

Option

/Zl

Ordinarily the compiler places the names of the default libraries (the standard C library plus the selected floating-point library or libraries) in the object file for the linker to read. This allows the default libraries to be linked with a program automatically.

The /Zl option suppresses the selection of default libraries. No library names are placed in the object file; as a result, the object file is slightly smaller.

The /Zl option is useful when you are building a library of routines. It is not necessary for every routine in the library to contain the default library information. Although the /Zl option saves only a small amount of space for a single object file, the total space savings is significant in a library containing many object modules. When you link a library of object modules created *with* the /Zl option with a C program file compiled *without* the /Zl option, the default library information is supplied by the program file.

Example

```
MSC ONE.C;
MSC /Zl TWO.C;
LINK ONE+TWO;
```

The first two commands create an object file named ONE.OBJ that contains the names of the standard C library (SLIBC.LIB) plus the emulator library and floating-point math library (EM.LIB and SLIBFP.LIB) and an object file named TWO.OBJ that contains no default library information. When ONE.OBJ and TWO.OBJ are linked, the default library information in ONE.OBJ causes the given libraries to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

7.7 Controlling Floating-Point Operations

By default, the compiler handles floating-point operations by using calls to an emulator library, which emulates the operation of an 8087 or 80287 coprocessor. If an 8087 or 80287 coprocessor is present at run time, it will be used. The floating-point (/FP) options give you a choice of five different methods of handling floating-point operations.

The advantages and disadvantages of each of the five /FP options are described in Section 3.7 of Chapter 3, "Compiling." You should read the discussion of floating-point options before reading this section. This section discusses two additional ways to control floating-point operations: by changing libraries at link time and by using the NO87 environment variable.

7.7.1 Changing Libraries at Link Time

When you compile using one of the floating-point options, the name of the corresponding library or libraries is placed in the object file for the linker to use. You can cause the linker to use a different floating-point library instead by using the /NOD (for no default library search) option at link time and specifying the name of a different library or libraries. The floating-point library names you can give on the link command line are

1. EM.LIB (the emulator) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model
2. 87.LIB (the 8087/80287 library), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model
3. SLIBFA.LIB, MLIBFA.LIB, or LLIBFA.LIB (the alternate math library), depending on the memory model

The 8087/80287 library (87.LIB) provides only minimal floating-point support. When you specify this library, an 8087 or 80287 coprocessor must be present at run time or the program will fail.

When you compile using the /FPa, /FPc, or /FPc87 option, you can specify any of the above libraries at link time. However, when you compile using the /FPi or /FPi87 option, you are not allowed to specify the alternate math library at link time; if you want to override the default library at link time, you must use either the emulator library or the 8087/80287 library, as appropriate.

When you use the /NOD option, the linker ignores all default library information in the object file. This means that you must give the name of the standard C library as well as the names of floating-point libraries at link time. Always give the name of the floating-point library or libraries *before* the name of the standard C library on the command line.

Examples

```
1.  MSC /AM CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,MLIBFA, MLIBC;

2.  MSC /FPa CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,EM.LIB+SLIBFP.LIB+SLIBC.LIB;

3.  MSC /FPi87 CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,EM.LIB+SLIBFP.LIB+SLIBC.LIB;
```

In the first example, the program CALC.C is compiled with the medium model option (/AM). No floating-point option is specified, so the default, /FPc, is used. /FPc generates floating-point calls and specifies the emulator (EM.LIB) plus MLIBFP.LIB in the object file. In the LINK command, the /NOD option is specified and the names of the alternate math library and the standard C library are given in the "Libraries" field. This causes any floating-point calls in CALC.OBJ, ANOTHER.OBJ, and SUM.OBJ to refer to the alternate math library instead of to the emulator. Notice that the medium model libraries (MLIBFA.LIB and MLIBC.LIB) must be used.

In the second example, CALC is compiled as small model (by default) and with the alternate math option (/FPa). The LINK command specifies the /NOD option and gives the names EM.LIB, SLIBFP.LIB, and SLIBC.LIB in the "Libraries" field, causing all floating-point calls to refer to the emulator library instead of the alternate math library.

In the third example, CALC.C is compiled with the /FPi87 option. The library names 87.LIB and SLIBFP.LIB are placed in the object file. The LINK command overrides the default library specification by giving the /NOD option and the names of the emulator library (EM.LIB), SLIBFP.LIB, and the standard library (SLIBC.LIB).

7.7.2 Using the NO87 Environment Variable

Programs compiled using the /FPc or /FPi option will automatically use an 8087/80287 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting an environment variable named NO87. (See Section 2.5 of Chapter 2, "Getting Started," or your MS-DOS documentation for a discussion of environment variables.)

If NO87 is currently set to any value when the program is executed, use of the 8087/80287 coprocessor is suppressed. The value of the NO87 setting is printed on the standard output as a message. The message is only printed if an 8087/80287 is present and suppressed; if no coprocessor is present, no message appears. If you don't want a message to be printed, set NO87 equal to one or more spaces; nothing will be printed.

Note that only the presence or absence of the NO87 definition is important in suppressing use of the coprocessor. The actual value of the NO87 setting is only used for printing the message.

The NO87 variable takes effect with any program linked with the emulator library (EM.LIB). It has no effect on programs linked with 87.LIB or SLIBFA.LIB (MLIBFA.LIB or LLIBFA.LIB for medium and large model programs).

Examples

```
1.  SET NO87=Use of coprocessor suppressed
2.  SET NO87=space
```

The first example causes the message "Use of coprocessor suppressed" to appear on the screen when a program that can use an 8087 or 80287 is executed.

The second example sets the NO87 variable to the space character. Use of the coprocessor is still suppressed, but no message is displayed.

7.8 Advanced Optimizing

This section describes additional optimizing procedures that can be used with the optimizing options described in Section 3.11 of Chapter 3, "Compiling," to create more efficient programs from your code.

7.8.1 Removing Stack Probes

Option

/Gs

You can reduce the size of a program and speed up execution slightly by using the /Gs option to remove all stack probes. A stack probe is a short routine called by a function to check the program stack for available space. The stack probe routine is called at every entry point. Ordinarily, the stack probe routine generates a message when it detects a stack overflow. When the /Gs option is used, no message is printed.

The /Gs option is useful when a program is known not to exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls.

Although the /Gs option, combined with the /Osa option, described with the O string options in Section 3.11 "Optimizing," makes the smallest possible program, it should be used with great care. Removing stack probes from a program means that some execution errors may not be detected.

Example

```
MSC FILE.C /Ota /Gs;
```

This example optimizes the file FILE.C by removing stack probes with the /Gs option and relaxing alias checking with the /Ota option. The letter "t" in the /Ota option tells the compiler to favor execution time over code size in the optimization.

7.8.2 Maximum Optimization

Option

`/Ox`

The /Ox option is a shorthand way to combine optimizing options to produce the smallest possible program. Its effect is equivalent to

`/Oas /Gs`

Thus, the /Ox option removes stack probes, relaxes alias checking, and favors code size over execution time.

7.9 Modifying the Executable File

The EXEPACK and EXEMOD utilities (supplied with your compiler software) allow you to modify an executable program file. The EXEPACK utility "compresses" the executable file by removing sequences of characters from the file and by optimizing the relocation table. Programs that have large data structures (arrays and **struct** types) contain sequences of null characters to initialize the data structures. By using EXEPACK on such program files, you can significantly reduce the size of the executable file and the time required for loading.

The EXEMOD utility allows you to examine and modify file header information. The program assumes that you are familiar with the fields and format of the file header. See your *Microsoft MS-DOS Programmer's Reference Manual* for details.

The following sections explain how to invoke and pass arguments to the EXEPACK and EXEMOD programs.

7.9.1 The EXEPACK Utility

Command

EXEPACK *executable-file output-file*

The EXEPACK utility compresses sequences of identical characters from the given *executable-file* and optimizes the relocation table. The compressed file is written to the output file, and the original file is unmodified.

The EXEPACK utility produces self-explanatory error messages if it is unable to compress the given file. See Section E.6 of Appendix E, "Error Messages," for a listing of these error messages.

7.9.2 The EXEMOD Utility

Command

EXEMOD *executable-file* [/stack *n*] [/min *n*] [/max *n*]

The EXEMOD utility modifies fields in the header according to instructions given on the command line. To display the header fields without modifying them, give the *executable-file* without any options.

The options are shown with the forward slash (/) option character, but a hyphen (-) may be used instead if you prefer. They can be given in either uppercase or lowercase. The options have the effects listed below.

Option	Effect
/stack *n*	Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary.
/min *n*	Sets the minimum allocation value to *n*, where *n* is a hexadecimal value in paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/max *n*	Sets the maximum allocation to *n*, where *n* is a hexadecimal value in paragraphs. The maximum allocation

value must be greater than or equal to the minimum allocation value.

Notice that the modifications you can make with the /stack and /max options can also be made by relinking the program with the corresponding linker options (/STACK and /CPARMAXALLOC). The advantage of the EXEMOD utility is that it modifies the executable file directly, without requiring the program to be relinked.

The EXEMOD program produces self-explanatory error messages when it is unable to carry out the given instructions. For a listing of these messages, see Section E.7 of Appendix E, "Error Messages."

Warning

The /stack option can only be used on programs compiled with the Microsoft C Compiler Version 3.0 or later, the Microsoft Pascal Compiler Version 3.3 or later, or the Microsoft FORTRAN Compiler Version 3.3 or later. Use of the /stack option with other programs may cause the program to fail.

7.10 Controlling Binary and Text Modes

Most C programs use one or more data files for input and output. Under MS-DOS, data files are ordinarily processed in "text" mode. In text mode, carriage return-linefeed combinations (CR-LF) are translated into a single linefeed (LF) character on input. Linefeed characters are translated to carriage return-linefeed combinations on output.

In some cases you may want to process files without making these translations. In binary mode, carriage return-linefeed translations are suppressed.

Standard library routines such as *fopen* or *open* give you the option to override the default mode when you open a particular file. You can also change the default mode for an entire program from text to binary mode. Do this by linking your program with the file BINMODE.OBJ, which is supplied as part of your C compiler software. Simply add the filename or pathname of BINMODE.OBJ to the list of filenames when you link your program.

When you link with BINMODE.OBJ, all files opened in your program default to binary mode, with the exception of *stdin*, *stdout*, and *stderr*. However, linking with BINMODE.OBJ does not force you to process all data files in binary mode. You still have the option to override the default mode when you open the file.

Use the *setmode* library function when you want to change the default mode of *stdin*, *stdout*, or *stderr* from text to binary, or the default mode of *stdaux* or *stdprn* from binary to text. The *setmode* function can change the current mode for any file and is primarily used for changing the mode of *stdin, stdout, stderr, stdaux,* and *stdprn.*

7.11 Mixed Model Programming

Option

/Astring

The Microsoft C Compiler defines three standard memory models (small, medium, and large) to accommodate programs with differing memory requirements. For an introduction to memory models, along with a discussion of the standard memory models and an alternate form of the /A option, see Section 3.13 of Chapter 3, "Compiling."

One limitation of the predefined memory model structure is that all pointers for code or data change size at once when you change memory models. To overcome this limitation, the Microsoft C Compiler lets you override the default addressing convention for a given memory model and access an item with either a "near" or a "far" pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program. You can access that item in another segment, saving space in your default data segment.

There are two ways to create mixed model programs. You can use the special keywords **near** and **far** in your program declarations to override the default addressing conventions for individual items. In a small model program, the **far** keyword lets you access data and functions in segments outside the program. In medium and large model programs, **near** lets you access data with just an offset. The **near** and **far** keywords can be used with a standard memory model to overcome addressing limitations for particular items without changing the addressing conventions for the program as a whole.

In the second method you combine features of the standard memory models to create your own customized memory model. To do this, you use the /A option followed by a string of three letters that tells the compiler the attributes of the memory model you want to use. The three fields of the string correspond to the code pointer size, the data pointer size, and the stack and data segment setup. The letters allowed in each field are unique, so you can give them in any order after /A. All three letters must be present.

Using the **near** and **far** keywords is the recommended procedure for creating mixed model programs. These keywords are particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program. You can use the **far** keyword to cause a new segment to be allocated for the data item, and then access that item with a far pointer, while still using near pointers (the default) for your other data. You must take care when calling library routines in programs that use the **near** and **far** keywords (for example, you cannot pass a **far** data item to a small model library routine), but in most cases you can easily use the standard libraries (small, medium, or large model) with these programs.

When you use a customized memory model (the /A*string* option), you are responsible for making sure that your program reflects the customized model and handles pointers and the stack and data segments consistently and correctly. Moreover, you are responsible for selecting and modifying a library for your program. Library support is not guaranteed for mixed model programs, and adapting one of the standard libraries (small, medium, or large model) to produce the appropriate combination of **near** and **far** routines and data may require considerable time and effort.

The sections below discuss the **near** and **far** keywords, pointer size, the stack and data segment setup, and library support for mixed model programs.

7.11.1 Using the Near and Far Keywords

The **near** and **far** keywords let you override the default addressing conventions of a particular memory model for an individual item (either code or data) without changing the default addressing conventions. The **near** and **far** keywords are special type modifiers you may use in your program declarations to override the default length and meaning of the address of a given variable. The **near** keyword defines an object with a 16-bit address. The **far** keyword defines an object with a full 32-bit segmented address. Any data item or function can be accessed with a **far** pointer.

When you use the **near** and **far** keywords in a program, you must specify the /Ze option at compile time to enable the keywords. (See Section 7.2, "Enabling Special Keywords.") Without the /Ze option, the compiler will treat **near** and **far** as ordinary identifiers, causing program errors.

Since there is no type-checking between items in separate source files, the **near** and **far** keywords should be used with great care.

The examples in the Table 7.1 illustrate the **far** and **near** keywords as used in declarations in a small model program.

Table 7.1

Uses of near and far Keywords with Small Model

Declaration	Address Size	Item Size
`char c;`	near (16 bits)	8 bits (data)
`char far d;`	far (32 bits)	8 bits (data)[a]
`char *p;`	near (16 bits)	16 bits (near pointer)
`char far *q;`	near (16 bits)	32 bits (far pointer)
`char * far r;`	far (32 bits)	16 bits (near pointer)[b]
`char far * far s;`	far (32 bits)	32 bits (far pointer)[c]
`int foo();`	near (16 bits)	function returning 16 bits
`int far foo();`	far (32 bits)	function returning 16 bits[d]

[a] This causes the variable to be allocated to a new segment.

[b] This example has no meaning; it is shown for syntactic completeness only.

[c] This is similar to accessing data in a long model program.

[d] This example leads to trouble in small model programs. In small model programs, calls to library functions are near. The far call shown in the example changes the CS (code segment) register, making the offset values used for library calls invalid.

The following example is from a medium model compilation.

```
int near foo();
```

This declaration prepares for a near function call in an otherwise far (calling) program.

7.11.2 Creating Customized Memory Models

The /A *string* option lets you change the attributes of the standard memory models to create your own memory models. With the /A *string* option you can control the default code pointer size, data pointer size, and the stack and data segment setup, as described below.

The standard memory model options (/AS, /AM, and /AL) can also be specified in the /A *string* form. The standard memory model options are listed with their /A *string* equivalents below.

/AS	/Asnd
/AM	/Alnd
/AL	/Alfd

7.11.2.1 Code Pointers

Options

/As*xx*
/Al*xx*

The letter "s" (for "short") tells the compiler to generate near (16-bit) pointers for all code items. This is the default for small model programs.

The letter "l" (for "long") means that far (32-bit) pointers are used to address all code items. Far pointers are the default for medium and large model programs.

The letters "l" and "s" are used for code pointers to distinguish them in the memory model string from the letters for data pointers, discussed below. The terms "short" and "long" are equivalent to "near" and "far", respectively.

7.11.2.2 Data Pointers

Options

/An*xx*
/Af *xx*

Two sizes are available for data pointers: near and far. The letter "n" tells the compiler to use near (16-bit) pointers for all data. This is the default for small and medium model programs.

The letter "f" specifies that all data pointers are far (32-bit). This is the default for large model programs. When far data pointers are used, no single data item may be larger than a segment (64K bytes) because address arithmetic is done only on 16 bits (the offset portion) of the address. All elements or members of the data item must be within the same segment to be referenced correctly.

7.11.2.3 Setting Up Segments

Options

/Ad*xx*
/Au*xx*
/Aw*xx*

The letter "d" tells the compiler that SS equals DS; that is, the stack and data segment pointers contain the same address. This is the default for all programs. In small and medium model programs, the stack and all data segments combined must occupy less than 64K bytes; thus, any data item is accessed with just 16-bit offset from the address in SS and DS.

In large model programs, global and static data are placed in a single segment called the "default" data segment. The address of this segment is stored in the DS and SS registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to keep in mind when passing pointers as arguments in large model programs.

The letter "u" allocates different segments for the stack and the data segments. Each object file (module) is allocated its own segment for global and static data items. When the "u" option is specified, the address in the DS register is saved at the entry point for each program module, and the new DS value for the module is loaded into the register. The previous DS value is restored at the module exit point. Thus, only one data segment is accessible at any given time.

A single segment is allocated for the stack, and its address is stored in the stack register. The stack cannot be placed in a data segment since it must be available throughout the entire program.

The "w" option sets up a separate stack segment, like the "u" option, but does not automatically load the DS register at each module entry point. This option is typically used when writing application programs that interface with an operating system or with a program running at the operating system level. The operating system or the program running beneath the operating system actually receives the data intended for the application program and places it in a segment; then it must load the DS register with the segment address for the application program.

7.11.3 Library Support

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the three standard memory models (small, medium, and large) through three separate run-time libraries. When you write mixed model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

When using the **near** and **far** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, medium, or large) with your program. However, you must take care when calling library routines; for example, you cannot pass **far** data items to a small model library routine.

Library support for programs using a customized memory model (the /A*string* option) is not guaranteed, and you will probably need to spend some time creating a customized library to be used with your customized memory model. It is recommended that you use the /NOD (for no default library search) option when linking, and then explicitly specify the library files and object files you want to use. Be sure to use the correct startup routine for your memory model; for example, if the source file containing the "main" function is compiled with far code pointers and near data

pointers as the default, you should use the startup file from the medium model library.

Moreover, you must make sure that any library routines called in those files are taken from the appropriate library. For example, suppose you compile the "main" function with the small memory model option (/AS) and compile another source file from the same program with the option /Alnd (which makes all code pointers far pointers). If the second program file contains calls to library routines, you can create a customized library file (using the LIB utility) that replaces each routine called in the second file with the corresponding large model version.

Notice that you must declare these large model routines appropriately as well. This procedure requires great caution, however, because the replaced routines may be called in other source files of the program as well. Moreover, some routines in the run-time library call other run-time routines to accomplish their tasks.

7.12 Setting the Data Threshold

Options

`/Gt[`*n*`]`

By default, the compiler allocates all static and global data items to the default data segment. The /Gt option causes all data items greater than *n* bytes to be allocated to a new data segment. When *n* is specified, it must follow the /Gt option immediately, with no intervening spaces. When *n* is omitted, the default threshold value is 256.

You can only use the /Gt option with large model programs since small and medium model programs have only one data segment. It is particularly useful with programs that have more than 64K bytes of static and global data in small data items.

7.13 Naming Modules and Segments

Options

/NM*module-name*
/NT*text-segment-name*
/ND*data-segment-name*

"Module" is another name for an object file created by the C compiler. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source filename. You can change this name using the /NM (for "name module") option. The new *module-name* can be any combination of letters and digits.

A "segment" is a contiguous block of binary information (code or data) produced by the C compiler. Every module has at least two segments: a text segment containing the program instructions and a data segment containing the program data. Each segment in every module has a name. This name is used by the linker to define the order in which the segments of the program appear in memory when loaded for execution. (Note that the segments in the group named DGROUP are an exception; see Section 8.1.1.1, "Segments," in Chapter 8, "Interfaces with Other Languages," for details.)

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small model programs the text segment is named "_TEXT" and the data segment is named "_DATA". These names are the same for all small model modules, so all text segments from all modules are loaded as one contiguous block, and all data segments from all modules form another contiguous block.

In medium model programs, the text from each module is placed in a separate segment with a distinct name, formed by using the module basename along with the suffix "_TEXT". The data segment is named "_DATA", as in the small model.

In large model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment is given the name of the module plus the suffix "_TEXT". The data from each segment are placed in a private segment with a unique name (except for global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 7.2.

Table 7.2
Segment Naming Conventions

Model	Text	Data	Module
Small	_TEXT	_DATA	*filename*
Medium	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA[a]	*filename*

[a] Name of default data segment.

You can override the default names used by the C compiler (thus overriding the default loading order) by using the /NT (for "name text") and /ND (for "name data") options. These options set the names of the text and data segments, in each module being compiled, to a given name. The *text-segment-name* and *data-segment-name* can be any combination of letters and digits.

Chapter 8
Interfaces with Other Languages

8.1 Assembly Language Interface 159
8.1.1 Segment Model 159
8.1.1.1 Segments 159
8.1.1.2 Groups 162
8.1.1.3 Classes 163
8.1.2 The C Calling Sequence 165
8.1.3 Entering an Assembly Routine 165
8.1.4 Return Values 166
8.1.5 Exiting a Routine 167
8.1.6 Naming Conventions 168
8.1.7 Register Considerations 169
8.1.8 Program Example 169
8.2 Calling FORTRAN and Pascal Routines 170

8.1 Assembly Language Interface

This section explains how to use 8086/8088 assembly language routines with C language programs and functions. In particular, it outlines the segment model used by the Microsoft C compiler and explains how to call assembly language routines from C language programs and vice versa. This assembly language interface is especially useful for those assembly language programmers who want to use the functions of the standard C library and other C libraries.

If you have assembly language programs that were written to work with the Microsoft C Compiler Version 2.03 or earlier, turn to Section D.4 of Appendix D, "Converting from Previous Versions of the Compiler," for a discussion of differences between the assembly language interface for this compiler and earlier versions.

8.1.1 Segment Model

This section describes the run-time structure of Microsoft C programs. Memory on the 8086/8088 processor is divided into segments, each containing up to 64K bytes. When a program is linked, the segments are organized into groups and classes. The segments, groups, and classes of Microsoft C programs are described below.

8.1.1.1 Segments

Figure 8.1 shows the order of primary segments of a C program in memory, from the highest memory location to the lowest. When you look at a map file produced by linking a C program, you may notice other segments in addition to the names listed below. These additional segments have specialized uses for Microsoft languages and should not be used by other programs.

The /DOSSEG option available with Microsoft LINK produces the ordering shown here. Since this is the default ordering for C programs, you do not need to use /DOSSEG with C programs, but you may find it useful when linking assembly language routines.

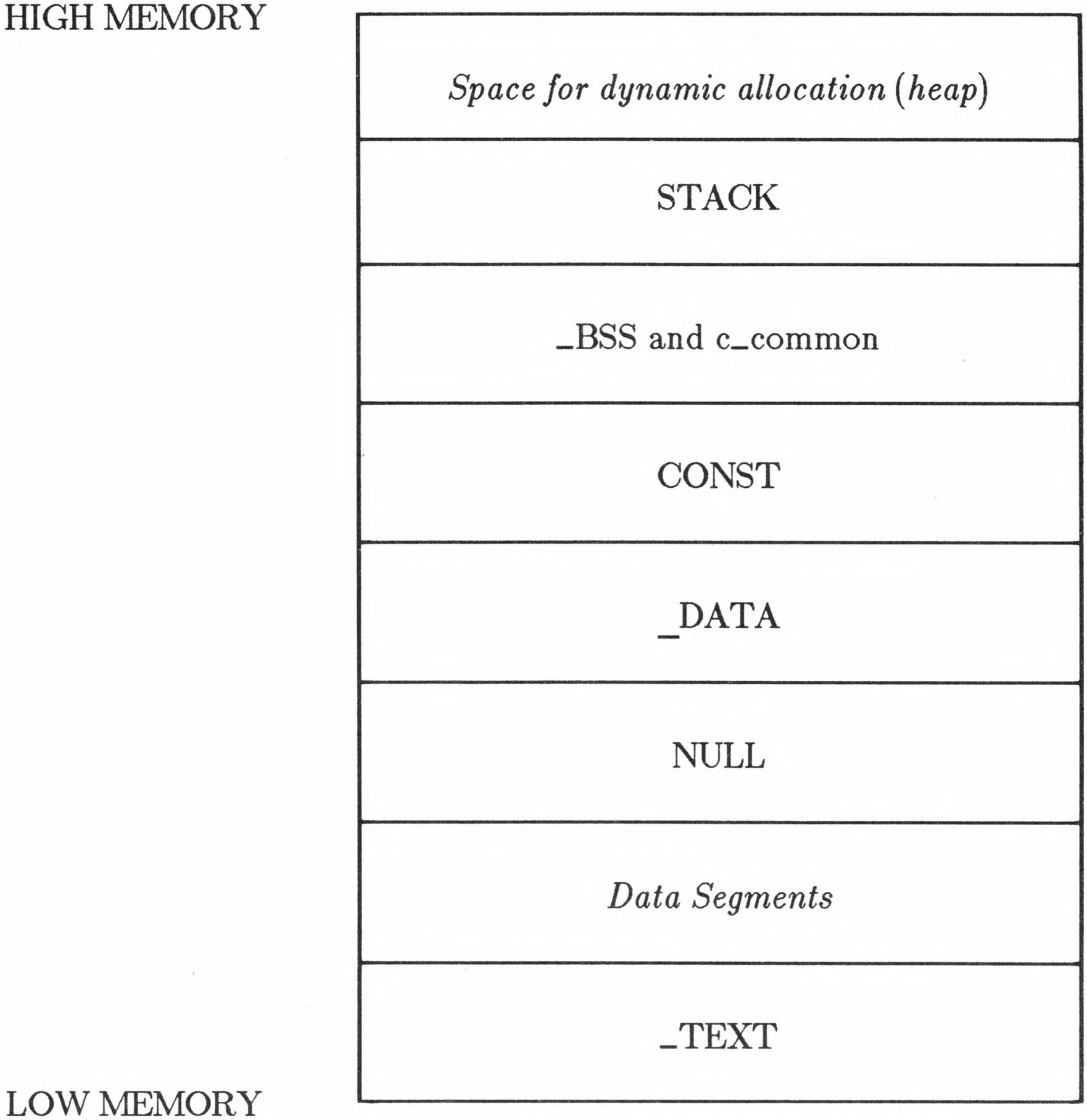

Figure 8.1 Segment Setup in C Programs

The "heap" is the area of unallocated memory that is available for dynamic allocation by the program. Its size varies, depending on the program's other storage requirements.

The segment contents are listed below.

STACK	The STACK segment contains the user's stack, which is used for all local data items.

_BSS

The _BSS segment contains all uninitialized static data items except for those that are explicitly declared as **far** items in the source file.

c_common

The c_common segment contains all uninitialized global data items for small and medium model programs. In large model programs, this type of data item is placed in a data segment with class FAR_BSS.

CONST

The CONST segment contains all constants that can only be read. These include floating-point constants, as well as segment values for data items declared **far** in the source file or data items that are forced into their own segment by use of the /Gt option.

Writing to string literals is allowed in C. Thus, strings are stored in the _DATA segment rather than the CONST segment.

_DATA

The _DATA segment is the default data segment. All initialized global and static data reside in this segment for all memory models, except for data explicitly declared **far** or data forced into different segments by use of the /Gt option.

NULL

The NULL segment is a special purpose segment that occurs at the beginning of DGROUP. The NULL segment contains the compiler copyright notice. This segment is checked before and after the program executes. If the contents of the NULL segment change in the course of program execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. The error message "Null pointer assignment" is displayed to notify the user.

Data Segments

Initialized static and global **far** data items are always placed in their own segments with class name FAR_DATA. This allows the linker to combine these items so that they all come before DGROUP. Uninitialized static and global **far** data items are placed in segments that have class FAR_BSS. Again, this allows the linker to place these items between the TEXT segment or segments and DGROUP. In large model programs global uninitialized data are treated as though

declared as **far** (unless specifically declared **near**) and given class FAR_BSS.

_TEXT — The _TEXT segment is the code segment. In small model programs the code for all modules is combined in this segment. In medium and large model programs each module is allocated its own text segment. The segments are not combined, so there are multiple text segments in medium and large model programs. Each segment in a medium or large model program is given the name of the module plus the suffix _TEXT.

When implementing an assembly language routine to call or be called from a C program, you will probably refer to the _TEXT and _DATA segments most frequently. The code for the assembly language routine should be placed in the _TEXT segment (or *module-name*_TEXT for medium and large model programs). Data should be placed in whichever segment is appropriate to their use, as described above. Usually this is the default data segment, _DATA.

8.1.1.2 Groups

All segments with the same group name must fit into a single physical segment, which is up to 64K bytes long. This allows all segments in a group to be accessed through the same segment register. The Microsoft C compiler defines one group named DGROUP.

The NULL, _DATA, CONST, BSS, c_common, and STACK segments are grouped together in the data group, called DGROUP. This allows the compiler to generate code for accessing data in each of these segments without constantly loading the segment values or using many segment overrides on instructions. DGROUP is addressed using the DS or SS segment register. DS and SS always contain the same value except when the "u" or "w" option of the /A option is used.

In large model programs, or small and medium model programs using **far** data declarations, DS may be changed temporarily to a different value to allow the program to access data outside the default data segment. The ES register may also be used in these cases.

SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. This provides compatibility with the Intel 80286 processor.

In small model programs, there is only one text segment, named _TEXT. In medium and large model programs, the names of all text segments must end with the suffix "_TEXT". The text segments are not grouped.

8.1.1.3 Classes

Table 8.1 gives the align type, combine class, class name, and group for each segment discussed above. All segments with the same class name are loaded next to each other.

Table 8.1

Segments, Groups, and Classes for Standard Memory Models

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
Small	_TEXT	byte	public	CODE	
	Data Segments[a]	para	private	FAR_DATA	
	Data Segments[b]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Medium	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data Segments[a]	para	private	FAR_DATA	
	Data Segments[b]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Large	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data Segments[c]	para	private	FAR_DATA	
	Data Segments[d]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

[a] Segment(s) for initialized **far** data.

[b] Segment(s) for uninitialized **far** data.

[c] Segment(s) for initialized global and static data.

[d] Segment(s) for uninitialized global and static data.

8.1.2 The C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembly language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char**, **short**, **int**, **unsigned char**, **unsigned short**, or **unsigned int** type occupy a single word (16 bits) on the stack. Arguments with **long** or **unsigned long** type occupy a double word (32 bits); the value's high order word is pushed first. Arguments with **float** type are converted to **double** type (64 bits). Note that **char** type arguments are sign-extended to **int** type before being pushed on the stack; **unsigned char** type arguments are zero-extended to **unsigned int** type.

Pointers occupy either 16 or 32 bits, depending on the memory model, the type of item addressed (code or data), and whether the pointer is modified with a **near** or **far** declaration. The segment value of far pointers is pushed first, then the offset.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed. Arrays are passed by reference; the array identifier evaluates to the array address, which is used to access the array.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

8.1.3 Entering an Assembly Routine

Assembly language routines that receive control from C function calls should preserve the contents of the BP, SI, and DI registers and set the BP register to the current SP register value before proceeding with their tasks. (It is not necessary to preserve the contents of SI and DI if the assembly language routine does not modify them.)

If the assembly routine modifies the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, their values should be saved on entry and restored at exit. The values of SS and DS are always equal in C programs unless the "u" or "w" letter of the /A option is

specified to set up separate stack and data segments.

The following example illustrates the recommended instruction sequence for entry to an assembly language routine.

```
entry:
        push    bp
        mov     bp,sp
        push    di
        push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument pushed by the function call (which is also the first argument given in the call's argument list) is at address [bp+4] for a near function call, [bp+6] for a far function call. Subsequent arguments begin at [bp+6], [bp+8], or [bp+10], depending upon the size of the first argument and whether the function call is near or far. If the first argument is a single word and the function call is near, the next argument starts at [bp+6]. If the first argument is a single word and the function call is far, or the first argument is a double word and the function call is near, the next argument starts at [bp+8]. If the first argument is a double word and the function call is far, the next argument starts at [bp+10].

Notice that the push instructions in the above sequence are not necessary if the assembly language routine does not modify the contents of the SI and DI registers, which are used by the compiler to store **register** variables.

It is recommended that macros be written and used to distinguish between near and far function calls and returns. Such macros make the code more readable and can help to insulate a program from changes in the calling sequence.

8.1.4 Return Values

Assembly language routines that wish to return values to a C language program or receive return values from C functions must follow the C return value conventions. The conventions are shown in Table 8.2.

Table 8.2
C Return Value Conventions

Return Value Type	Register
char	AX
short	AX
int	AX
unsigned char	AX
unsigned short	AX
unsigned int	AX
long	high order word in DX; low order word in AX
unsigned long	high order word in DX; low order word in AX
struct or **union**	address of value in AX; value must be in a static area in memory
float or **double**	address of value in AX; value must be in a static area in memory
near pointer	AX
far pointer	segment selector in DX; offset in AX

8.1.5 Exiting a Routine

Assembly language routines that return control to C programs should restore the values of the BP, SI, and DI registers before returning control. (The contents of the SI and DI registers do not have to be restored if the entry sequence did not push them.) The following example illustrates the recommended instruction sequence for exiting a routine called by a small model program.

```
pop     si
pop     di
mov     sp,bp
pop     bp
ret
```

This sequence does not change the AX, BX, CX, or DX registers or any of the segment registers. The sequence does not remove arguments from the stack; this is the responsibility of the calling routine.

Notice that the "pop" instructions for SI and DI in the above sequence are not necessary if the contents of the SI and DI registers are not modified by the assembly language routine and were not saved on entry.

8.1.6 Naming Conventions

An assembly language routine can access globally visible items (data or functions) in a C program by prefixing the item name with an underscore (_). (C items declared as **static** cannot be accessed.) For example, a C function named *add* can be accessed in an assembly language program by declaring the name *_add* as external.

For a C program to access an assembly language routine or data item, the name of the assembly language item must begin with an underscore (_). The C program refers to the assembly language item without the underscore. For example, a C program could call a publicly defined assembly language routine named *_mix* by declaring

```
extern mix();
```

If the assembly language name does not begin with an underscore, it cannot be accessed in a C program.

The C compiler reserves some identifiers beginning with two underscores for internal use. You should avoid using identifiers with two leading underscores in your assembly routines and identifiers with one leading underscore in your C source files, as these identifiers may conflict with internal names.

Some assemblers translate all lowercase letters to uppercase, or vice versa. Since the C language is case-sensitive, this can pose problems. Check your assembler documentation for information on this topic. The Microsoft Macro Assembler, Version 3.0 or later, offers an option to control case sensitivity.

8.1.7 Register Considerations

The SI and DI registers are used to store the values of variables given **register** storage in a C program. An assembly language routine that changes the SI and DI registers is responsible for saving their contents on entry and restoring them before exiting.

The C compiler assumes that the direction flag is always cleared. If your assembly routine sets the direction flag, be sure to clear it (using the "cld" instruction) before returning.

If the assembly routine modifies the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, their values should be saved on entry and restored at exit. The values of SS and DS are always equal in C programs unless the "u" or "w" letter of the /A option is specified to set up separate stack and data segments.

8.1.8 Program Example

To illustrate the assembly language interface, consider the following example. The example assumes that the C program is small model.

```
int a=1, b=1, c;

main()
{
        c = add(a,b);
}

add(i,j)
int i,j;
{
        return(i+j);
}
```

If the *add* function is written as an assembly language routine instead of a C function, it must save the proper registers; retrieve the arguments from the stack; add the arguments; place the return value in the AX register; and then restore registers and return control. Here is an example of how the routine can be written. Notice that preserving and restoring SI and DI is shown for illustration, although the procedure is not strictly necessary in this case. (If the assembly routine were written to work with a medium or large model C program, the *_add* procedure would be declared FAR instead of NEAR.)

```
; i = [bp + 4]
; j = [bp + 6]

_add    PROC NEAR
        push    bp
        mov     bp,sp
        push    di
        push    si

        mov     ax,[bp+4]
        add     ax,[bp+6]

        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret

_add endP
```

If, on the other hand, the C function is to be called by an assembly language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the AX register. These instructions are shown in the following example.

```
push    [_b]
push    [_a]
call    _add
add     sp,4
mov     [_c],ax
```

8.2 Calling FORTRAN and Pascal Routines

This section describes how to use the **fortran** and **pascal** keywords in C programs to declare routines written in Microsoft FORTRAN or Microsoft Pascal. To call such routines, you must have Version 3.3 or later of the Microsoft FORTRAN or Microsoft Pascal compilers.

Writing mixed-language programs requires that you have a thorough understanding of the languages involved. For example, you must consider

- the correspondence between data types in different languages
- the rules for converting, passing, and returning values in different languages
- naming conventions
- library support and compatibility between libraries
- the memory models available in each language

This section discusses only the syntax of **pascal** and **fortran** declarations and does not attempt to cover the many issues that arise in mixed language programming. For a complete discussion of mixed language programming, see your Pascal or FORTRAN manual, Version 3.3 or later.

The special keywords **fortran** and **pascal** let you declare external FORTRAN and Pascal routines in C programs. To enable the keywords, you must use the /Ze option when compiling.

You can also use the **fortran** and **pascal** keywords when declaring a C function. In this case the keyword instructs the C compiler to use FORTRAN or Pascal conventions for entry and exit sequences when compiling the C function. This option is useful when you want to call a C function from within a FORTRAN or Pascal routine.

You declare FORTRAN and Pascal routines in the same manner that you declare C functions: you specify the function identifier, the return type, and the type and number of arguments to the function. (See the *Microsoft C Language Reference* for a complete discussion of the syntax of function declarations.)

The following additional rules apply to **fortran** and **pascal** declarations.

1. Whenever a **fortran** or **pascal** keyword is used in a declaration, the types of the arguments must be declared with an argument type list. (Ordinarily the argument type list is optional.)
2. The **fortran** and **pascal** keywords modify the item immediately to the right in a declaration.
3. The special **near** and **far** keywords can be used with the **fortran** and **pascal** keywords in declarations. The sequences **far fortran** and **fortran far** are equivalent.

Complex declarators are allowed in **pascal** and **fortran** declarations, just as in C function declarations. With the addition of the **near** and **far** modifiers, these declarations can become quite long. The following

examples illustrate the syntax of **pascal** and **fortran** declarations. For more on complex declarators, see the *Microsoft C Language Reference.*

Examples

```
1.  short pascal thing(short, short);

2.  long (pascal *thing)(void);

3.  short (pascal *(*thing)(void))(long);

4.  short near pascal thing(short);

5.  short pascal near thing(short);
```

Example 1 declares *thing* to be a Pascal routine taking two **short** arguments and returning a **short** value.

In Example 2, *thing* is declared as a pointer to a Pascal routine that takes no arguments and returns a **long** value.

Example 3 declares *thing* to be a pointer to a C function taking no arguments that returns a pointer to a Pascal routine. The Pascal routine is declared to take one **long** argument and return a **short** value.

Examples 4 and 5 are equivalent. Both declare *thing* to be a **near** Pascal routine. The routine takes one **short** argument and returns a **short** value.

Appendices

User's Guide Appendices

A ASCII Character Codes 175

B Command Summary 177

C The CL Command 189

D Converting from Previous Versions of the Compiler 199

E Error Messages 223

F Working with Microsoft Products 265

G Microsoft LINK Technical Summary 273

Appendix A
ASCII Character Codes

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
000	000	00H	NUL	032	040	20H	SP
001	001	01H	SOH	033	041	21H	!
002	002	02H	STX	034	042	22H	"
003	003	03H	ETX	035	043	23H	#
004	004	04H	EOT	036	044	24H	$
005	005	05H	ENQ	037	045	25H	%
006	006	06H	ACK	038	046	26H	&
007	007	07H	BEL	039	047	27H	'
008	010	08H	BS	040	050	28H	(
009	011	09H	HT	041	051	29H	)
010	012	0AH	LF	042	052	2AH	*
011	013	0BH	VT	043	053	2BH	+
012	014	0CH	FF	044	054	2CH	,
013	015	0DH	CR	045	055	2DH	-
014	016	0EH	SO	046	056	2EH	.
015	017	0FH	SI	047	057	2FH	/
016	020	10H	DLE	048	060	30H	0
017	021	11H	DC1	049	061	31H	1
018	022	12H	DC2	050	062	32H	2
019	023	13H	DC3	051	063	33H	3
020	024	14H	DC4	052	064	34H	4
021	025	15H	NAK	053	065	35H	5
022	026	16H	SYN	054	066	36H	6
023	027	17H	ETB	055	067	37H	7
024	030	18H	CAN	056	070	38H	8
025	031	19H	EM	057	071	39H	9
026	032	1AH	SUB	058	072	3AH	:
027	033	1BH	ESC	059	073	3BH	;
028	034	1CH	FS	060	074	3CH	<
029	035	1DH	GS	061	075	3DH	=
030	036	1EH	RS	062	076	3EH	>
031	037	1FH	US	063	077	3FH	?

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character
LF=Line Feed, FF=Form Feed, CR=Carriage Return
DEL=Rubout

Appendix A *(continued)*

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
064	100	40H	@	096	140	60H	`
065	101	41H	A	097	141	61H	a
066	102	42H	B	098	142	62H	b
067	103	43H	C	099	143	63H	c
068	104	44H	D	100	144	64H	d
069	105	45H	E	101	145	65H	e
070	106	46H	F	102	146	66H	f
071	107	47H	G	103	147	67H	g
072	110	48H	H	104	150	68H	h
073	111	49H	I	105	151	69H	i
074	112	4AH	J	106	152	6AH	j
075	113	4BH	K	107	153	6BH	k
076	114	4CH	L	108	154	6CH	l
077	115	4DH	M	109	155	6DH	m
078	116	4EH	N	110	156	6EH	n
079	117	4FH	O	111	157	6FH	o
080	120	50H	P	112	160	70H	p
081	121	51H	Q	113	161	71H	q
082	122	52H	R	114	162	72H	r
083	123	53H	S	115	163	73H	s
084	124	54H	T	116	164	74H	t
085	125	55H	U	117	165	75H	u
086	126	56H	V	118	166	76H	v
087	127	57H	W	119	167	77H	w
088	130	58H	X	120	170	78H	x
089	131	59H	Y	121	171	79H	y
090	132	5AH	Z	122	172	7AH	z
091	133	5BH	[	123	173	7BH	{
092	134	5CH	\	124	174	7CH	|
093	135	5DH	]	125	175	7DH	}
094	136	5EH	^	126	176	7EH	~
095	137	5FH	_	127	177	7FH	DEL

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character
LF=Line Feed, FF=Form Feed, CR=Carriage Return
DEL=Rubout

Appendix B
Command Summary

B.1 Introduction 179
B.2 Compiler Summary 179
B.2.1 MSC Options 180
B.2.2 Standard Memory Models 183
B.2.3 Pointer and Integer Sizes 183
B.2.4 Segment Names 184
B.3 Linker Summary 184
B.3.1 Linker Command Characters 184
B.3.2 Linker Options 185
B.4 The LIB Utility 187
B.5 The EXEPACK Utility 187
B.6 The EXEMOD Utility 188

B.1 Introduction

This appendix summarizes the commands and options available with MSC and the following Microsoft utilities: LINK, LIB, EXEPACK, and EXEMOD.

B.2 Compiler Summary

The compiler is invoked with the MSC command. Type MSC to be prompted for responses or use the command line method to give information to MSC. If you don't give MSC all the information it needs on the command line, it will prompt you for the remaining responses.

Options can appear anywhere a space can appear. The options available with MSC are summarized below.

MSC uses three environment variables to locate the files it needs. Before invoking MSC, use the MS-DOS commands PATH and SET to assign a pathname or pathnames to the following variables.

PATH	Executable compiler files
INCLUDE	Include files
TMP	Temporary files

B.2.1 MSC Options

The following is a complete list of MSC options in alphabetical order. The hyphen character (-) can be used in place of the forward slash (/) to introduce the option if you prefer. Some additional options are available with the CL command; see Section C.4 of Appendix C, "The CL Command."

Option	Task
/A*letter*	Sets the program configuration. The *letter* may be S, M, or L, standing for "small," "medium," and "large" model, respectively.
/A*string*	Sets the program configuration. The *string* consists of three characters in any order, one from each of the following groups. Code Pointer Size: s small; l large Data Pointer Size: n near; f far Segment Setup: d SS equal to DS; u SS not equal to DS, DS loaded for each module; w SS not equal to DS, DS fixed
/C	Preserves comments when preprocessing a file (use only with /E, /P, or /EP).
/D*identifier* [= [*string*]]	Defines *identifier* to the preprocessor. The value is *string* or empty.
/E	Preprocesses the source file, copying the result to the standard output and inserting **#line** directives.
/EP	Preprocesses the source file, copying the result to the standard output without **#line** directives.
/Fa [*filename*]	Produces assembly listing.
/Fc [*filename*]	Produces combined source-assembly listing.

/Fl [*filename*]	Produces object listing.
/Fo*filename*	Names the object file.
/FPa	Generates floating-point calls and selects alternate math library.
/FPc	Generates floating-point calls and selects emulator (uses 8087/80287 if one is present).
/FPc87	Generates floating-point calls and selects 8087/80287 library (requires an 8087 or 80287 at run time).
/FPi	Generates in-line 8087/80287 instructions and selects emulator (uses 8087 or 80287 if one is present).
/FPi87	Generates in-line 8087/80287 instructions and selects 8087/80287 library (requires an 8087 or 80287 at run time).
/G0	Generates 8086/8088 instructions.
/G1	Generates 80186/80188 instructions.
/G2	Generates 80286 instructions.
/Gs	Removes calls to stack probe routine.
/Gt [*number*]	Places data items greater than *number* bytes in new segment (256 bytes is the default).
/H*number*	Restricts significant characters of external names to *number* characters.
/I*directory*	Adds *directory* to the top of the list of directories to be searched for include files.
/ND*data-segment-name*	Sets the data segment name.
/NM*module-name*	Sets the module name.
/NT*text-segment-name*	Sets the text segment name.

/O*string*	Controls optimization. The *string* consists of one or more of the following characters. d disable optimization a relax alias checking s favor code size t favor execution time x maximum optimization (equivalent to /Oas /Gs)
/P	Preprocesses the source file and sends output to file with the basename of the source file and the extension ".I".
/u	Removes definitions of all four predefined identifiers.
/U*identifier*	Removes definition of the given predefined identifier.
/V*string*	Copies *string* to the object file.
/w	Suppresses compiler warning messages.
/W*n*	Sets the output level ($n = 0, 1, 2$, or 3) for compiler warning messages.
/X	Ignores the list of "standard places" in the search for include files.
/Zd	Includes line number information in object file.
/Ze	Enables language extensions **far**, **fortran**, **huge**, **near**, and **pascal**.
/Zg	Generates function declarations from function definitions and writes declarations to standard output.
/Zl	Removes default library information from object file.
/Zp	Packs structure members.
/Zs	Performs syntax check only.

B.2.2 Standard Memory Models

Table B.1 defines the number of text and data segments for small, medium, and large memory models.

Table B.1

Text and Data Segments in Standard Memory Models

Model	Text Segments	Data Segments
Small	1	1
Medium	1 per module	1
Large	1 per module	1 default data segment[a]

[a] The number of additional data segments depends on the program requirements.

B.2.3 Pointer and Integer Sizes

Table B.2 defines the sizes (in bits) of data pointers, text pointers, and integers (`int` type) in the three standard memory models.

Table B.2

Pointer and Integer Sizes in Standard Memory Models

Model	Data Pointer	Text Pointer	Integer
Small	16	16	16
Medium	16	32	16
Large	32	32	16

B.2.4 Segment Names

The Table B.3 lists the default text and data segment names in the standard memory models. The default *modulename* is the filename.

Table B.3

Segment Names in Standard Memory Models

Model	Text	Data
Small	_TEXT	_DATA
Medium	*modulename*_TEXT	_DATA
Large	*modulename*_TEXT	_DATA[a]

[a] Name of default data segment; other data segments have unique, private names.

B.3 Linker Summary

Microsoft LINK, the object code linker utility, recognizes the command characters and options listed in this section. LINK uses the environment variable LIB to locate library files. Before invoking LINK, use the MS-DOS command SET to assign a pathname or pathnames to the LIB variable.

B.3.1 Linker Command Characters

Character	Task
+	Use the plus sign (+) to separate entries and to extend the current line in response to the "Object Modules" and "Libraries" prompts.
;	To select default responses to the remaining prompts, use a single semicolon (;) followed immediately by a RETURN any time after the first prompt.
CONTROL-C	Type CONTROL-C to terminate the link session at any time.

B.3.2 Linker Options

Options control various linker functions. Options must be typed at the end of a prompt response, regardless of which method is used to start Microsoft LINK. Options may be grouped at the end of any response, or may be scattered at the end of several responses. If more than one option is typed at the end of a response, each option must be preceded by a forward slash (/).

All options may be abbreviated. The only restrictions are that an abbreviation must be sequential from the first letter through the last typed and must uniquely identify the option.

Some linker options take numerical arguments. A numerical argument can be any of the following.

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with a zero. For example, the number "10" is a decimal number, but the number "010" is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with "0x". For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

The linker options, summarized below, are listed in alphabetical order.

Option **Task**

/CPARMAXALLOC:*number*
Sets the cparmaxALLOC field at offset 0x0c in the EXE header to *number*.

/DOSSEG Enforces the following loading order.

1. All segments with a class name ending with CODE.
2. All other segments outside of DGROUP.
3. DGROUP segments, in this order: (a) any segments of class BEGDATA (this class name is reserved for Microsoft use); (b) any segments not of class BEGDATA, BSS or STACK; (c) segments of class BSS; (d) segments of class STACK.

/DSALLOCATE
: Tells Microsoft LINK to load all data at the high end of the data segment. Do not use the /DSALLOCATE option with C programs.

/HIGH
: Causes the run file to be placed as high as possible in memory. Do not use the /HIGH option with C programs.

/LINENUMBERS
: Includes in the list file the line numbers and addresses of the source statements in the input modules.

/MAP
: Creates a file listing all public (global) symbols defined in the input modules.

/NODEFAULTLIBRARYSEARCH
: Causes default libraries to be ignored.

/NOGROUPASSOCIATION
: Provides compatibility with previous versions of LINK. Do not use the /NOGROUPASSOCIATION option with C programs.

/NOIGNORECASE
: Causes the linker to distinguish between uppercase and lowercase letters.

/OVERLAYINTERRUPT:*number*
: Sets the overlay interrupt number to *number*.

/PAUSE
: Causes Microsoft LINK to pause in the link session when the option is encountered.

/SEGMENTS:*number*
: Sets the number of segments the linker allows a program to have. The default is 128.

/STACK:*number*
: Sets the stack size to *number*, which may be any positive value up to 65,536 bytes. The default for C programs is 2K (2,048 bytes).

B.4 The LIB Utility

The following command characters are recognized by Microsoft LIB, the library manager utility.

Character	Task
+	Appends an object file or library file to the given library.
-	Deletes a module from the library.
-+	Replaces a module by deleting a module and appending an object file with the same name.
*	Extracts a module from the library and saves it in an object file.
-*	Extracts a module from the library and deletes it from the library after saving it in an object file.
;	Uses default responses to remaining prompts.
&	Extends current physical line; repeats command prompt.
CONTROL-C	Terminates library session.

B.5 The EXEPACK Utility

Command

EXEPACK *executable-file output-file*

The EXEPACK utility compresses sequences of identical characters from the given *executable-file* and optimizes the relocation table. The compressed file is written to the output file, and the original file is unmodified.

B.6 The EXEMOD Utility

Command

EXEMOD *executable-file* [/stack *n*] [/min *n*] [/max *n*]

The EXEMOD utility modifies fields in the header according to instructions given on the command line. To display the header fields without modifying them, give the *executable-file* without any options.

Option	Task
/stack *n*	Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary.
/min *n*	Sets the minimum allocation value to *n*, where *n* is a hexadecimal value in paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/max *n*	Sets the maximum allocation to *n*, where *n* is a hexadecimal value in paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value.

Appendix C
The CL Command

C.1 Introduction 191
C.2 Command Syntax and Options 191
C.3 Linking with the CL Command 195
C.4 Additional Options 196
C.5 XENIX-Compatible Options 196

C.1 Introduction

This appendix summarizes the CL command. The CL command can be used instead of the MSC and LINK commands to invoke the compiler and linker. It is similar to the "cc" interface used on XENIX and UNIX systems; therefore, it may be familiar to some users.

The CL command uses the same four environment variables used by the MSC and LINK commands. They are:

PATH
INCLUDE
TMP
LIB

C.2 Command Syntax and Options

The CL command has the form:

`CL [`*`options...`*`] `*`filename...`*` [/link `*`lib-field`*`]`

where each *option* is a command option, and each *filename* specifies a file to be processed. You can give more than one option or filename, but you must set off each item with one or more spaces. The /link option allows you to pass information to the linker; see Section C.3, "Linking with the CL Command," for a description of the /link option.

Each *filename* must be the name of a C language source file or an object file. If a source file, the filename must include the extension ".c" or ".C". When CL processes the file, it looks at the filename extension to determine whether it should start processing at the compiling or linking stage. Any files ending with ".c" or ".C" are compiled; files with any other extension or no extension are assumed to be object files.

You can use the MS-DOS "wild card" characters (? and *) in filenames on the CL command line. The CL command expands these characters in the same manner that MS-DOS does. See your MS-DOS documentation for more details.

An option consists of a forward slash (/) followed by a combination of one or more letters that have special meaning to CL. You can use a hyphen (-) instead of the slash as an option character if you prefer. You can use any of the options available with the MSC command with CL. Additional options that apply to CL only are given below.

Since you can process more than one file at a time with the CL command, the order in which you give listing options (the /F group of options) is important. The /Fa, /Fc, /Fl, and /Fo options available with the MSC command are also available with CL. In addition, you can use the /Fe option to name the executable file produced in the linking stage and the /Fm option to create a map file. The /F options that can be used with the CL command are summarized in Table C.1. Some additional rules that apply to arguments of the /F options when used with the CL command are given in Table C.2.

Table C.1
Summary of /F Options

Option	Task	Default Filename[a]	Default Extension
/Fa	Produces assembly listing	Basename of source file plus ".ASM"	.ASM
/Fc	Produces combined source-assembly listing	Basename of source file plus ".COD"	.COD
/Fe	Names the executable file	Basename of first source or object file on command line plus ".EXE"	.EXE
/Fl	Produces object listing	Basename of source file plus ".COD"	.COD
/Fm	Creates map file	Basename of first source or object file on the command line plus ".MAP"	.MAP
/Fo	Names object file	Basename of source file plus ".OBJ"	.OBJ

[a] The default filename for the /Fa, /Fc, /Fl, and /Fm options is used when the option is given with no argument or with a directory name as argument. The default filename for the /Fe and /Fo options is used when the option is not given at all, or when a directory name is given as the argument to the option.

Table C.2
Arguments to /F Options

Options	Filename Argument	Pathname Argument	No Argument
/Fa, /Fc, /Fl	Creates a listing for next source file on command line; uses default extension if no extension is supplied.	Creates listings in the given directory for every source file listed after the option on the command line; uses default names.	Creates listings in the default directory for every source file listed after the option on the command line; uses default names.
/Fe	Uses given filename for the executable file; uses default extension if no extension is supplied.	Creates executable file in the given directory; uses default name.	Not applicable; argument is required.
/Fm	Uses given filename for the map file; uses default extension if no extension is supplied.	Creates map file in the given directory; uses default name.	Uses default name.
/Fo	Uses given filename as the object filename for the next source file on command line; uses default extension if no extension is supplied.	Creates object files in the given directory for every source file listed after the option on the command line; uses default names.	Not applicable; argument is required.

Note: No spaces are allowed between the option and the argument (if any) for any of the /F options.

Unlike the MSC command, the CL command invokes the linker as well as the compiler. By default, CL automatically performs linking; you can override this with the /c option, described in Section C.4, "Additional Options." You can also pass your own arguments to the linker, in addition to the default arguments given by CL. This is described in Section C.3, "Linking with the CL Command."

C.3 Linking with the CL Command

By default, the CL command invokes the linker after compiling. You can override the default and cause CL to stop after compiling by giving the /c ("compile only") option.

The CL command uses the response file method of invoking the linker. By default, it builds the following response file.

```
LINK object-list /NOI
basename
NUL;
```

Notice that, by default, the "Libraries" field is not given. The names of the default libraries (the standard C library of the appropriate memory model, plus the appropriate floating-point library as determined by the floating-point option used) are encoded in the object file. The linker searches for the default libraries in the current working directory, then in the directories specified in the LIB environment variable, if any.

The *object-list* is a list of all object files produced in the compiling stage of the CL command, plus any object files specified on the CL command line. The /NOI option tells the linker *not* to ignore case; uppercase and lowercase letters are considered distinct. The *basename* is the name supplied for the executable file; it corresponds to the basename of the first source or object file on the CL command line. (However, you can provide a different name by using the /Fe option.) By default, no map file is produced, since the name NUL is provided in the third field. Notice, however, that the /Fm option can be used in the CL command to override the default and produce a map file.

You can supply your own responses for the "Libraries" field by using the /link option. This option, if included, must be the last item on the CL command line. Any libraries specified in the *library-field* are searched before the default libraries.

The *library-field* can contain one or more of the following.

1. *A pathname.* The linker searches the given pathname for the default libraries *before* searching directories given by the LIB variable.
2. *Additional or alternate library names.* If a pathname is included with the library name, only that pathname is searched. Otherwise,

the linker uses the standard library search path.

3. *The name of a floating-point library or libraries.* Any floating-point calls in your program refer to the given floating-point library instead of the default floating-point library.

4. *Options.* You can give any of the linker options described in Chapter 4, "Linking."

See Chapter 4, "Linking," for more details on default libraries (Sections 4.3.2 and 4.3.3); the library search path (Section 4.3.2); and linker options (Sections 4.3.4 and 4.5).

C.4 Additional Options

In addition to the MSC options summarized in Section B.2.1 of Appendix B, "Command Summary," the CL command recognizes the options listed below. The options are shown with the slash character (/) but can be given with the hyphen (-) character if you prefer.

Option	Task
/c	Creates an object file for each source file on the command line; suppresses linking.
/Fe*progname*	Names the executable program file with *progname*.
/Fm [*mapname*]	Creates a map file.
/link *library-field*	Passes the given *library-field* to the linker as the "Libraries" field.

C.5 XENIX-Compatible Options

To provide as much compatibility as possible with XENIX C compilers, the CL command also accepts the options recognized by the "cc" command on XENIX systems. Many of these options are identical to the MSC and CL options given in this manual; others have identical functions but different names. The complete list of XENIX options accepted by the CL command is given in Table C.3.

Table C.3
XENIX Options Accepted by the CL Command

XENIX Option	Task	MSC/CL Option
-c	Creates a linkable object file for each source file.	Same; CL only.
-C	Preserves comments when preprocessing a file (only when -P or -E).	Same.
-D name [=string]	Defines *name* to the preprocessor. The value is *string* or 1.	Same.
-E	Preprocesses each source file, copying the result to the standard output.	Same.
-F num	Sets the size of the program stack. The given size must be a hexadecimal number.	No equivalent in MSC, but /STACK option can be used with LINK or with the /link option of CL.
-I pathname	Adds *pathname* to the list of directories to be searched for **#include** files.	Same.
-K	Removes stack probes from a program.	-Gs
-L	Creates an object listing file containing assembled object code.	-Fl
-Mstring	Sets the program configuration. The *string* may be any combination of "s" (small model), "m" (middle model), "l" (large model), "e" (enable **far** and **near** keywords), "2" (enable 286 code generation), "b" (reverse word order), and "t" (set data threshold for largest item in a segment). The "s", "m", and "l" options are mutually exclusive.	-Me is equivalent to Ze. -M2 is equivalent to G2. -Mt is equivalent to Gt. -Mb has no equivalent. -Ms is equivalent to AS. -Mm is equivalent to AM. -Ml is equivalent to AL.
-nl	Sets the maximum length of external symbols.	-H

-ND name	Sets the data segment name.	Same.
-NM name	Sets the module name.	Same.
-NT name	Sets the text segment name.	Same.
-o filename	Makes *filename* the name of the final executable program.	-Fo
-O	Invokes the object code optimizer.	-Os (default)
-P	Preprocesses source files and sends output to files with the extension ".i".	Same.
-S	Creates an assembly source listing.	-Fa
-V string	Copies *string* to the object file.	Same.
-w	Suppresses compiler warning messages.	Same.
-W num	Sets the output level for compiler warning messages.	Same.
-X	Removes the standard directories from the list of directories to be searched for **#include** files.	Same.

Appendix D
Converting from Previous Versions of the Compiler

D.1 Introduction 201
D.2 Language Definition Differences 201
D.3 Run-Time Library Differences 206
D.3.1 abs 208
D.3.2 creat 208
D.3.3 fopen, freopen 209
D.3.4 iscsym, iscsymf 210
D.3.5 max 210
D.3.6 min 210
D.3.7 movmem 210
D.3.8 open 211
D.3.9 setmem 211
D.3.10 setnbuf 211
D.3.11 stcis, stcisn, stclen, stpbrk, stpchr, stscmp 212

D.4 Differences in Assembly Language Interface 212
D.4.1 Register Usage Conventions 213
D.4.2 Stack Setup and Subroutine Entry/Exit Code 214
D.4.3 Global Variable Naming Conventions 219
D.4.4 Segment Usage and Naming 219

D.1 Introduction

This appendix describes differences between Version 3.0 and earlier versions of the Microsoft C Compiler. The differences fall into three categories: language definition differences, run-time library differences, and assembly language interface differences.

The changes in Version 3.0 are designed to conform to the ANSI standard for the C language (still under development) and to the original language definition. Some features of Versions 2.03 and earlier were not compatible with these standards; such features have been eliminated or changed in Version 3.0. The changes in Version 3.0 also provide greater portability of source code, particularly in the run-time library.

An include file, V2TOV3.H, accompanies your compiler software to help you run your existing Microsoft C programs under Version 3.0.

The following sections describe language, run-time library, and assembly language differences in detail, and outline strategies for converting existing programs.

D.2 Language Definition Differences

This section lists differences in the definition of the C language between Versions 3.0 and earlier versions. The differences are listed by section number from Appendix A of *The C Programming Language,* by Brian Kernighan and Dennis Ritchie, published in 1978 by Prentice-Hall.

Section Number Kernighan and Ritchie	Differences in Versions of Microsoft C
2.1	Comments do not nest in Version 3.0. Versions 2.03 and earlier permitted nesting of comments, unless nesting was deliberately turned off with a command line option. Code containing nested comments will not compile correctly under Version 3.0. In Version 3.0 you can suppress compilation of program sections that contain comments by using a

preprocessor directive (**#if**).

2.2 Under Version 3.0, identifiers must begin with a letter of the alphabet (uppercase or lowercase) or the underscore character (_). The same characters plus the digits 0-9 are allowed for the rest of the identifier. Versions 2.03 and earlier also allow the dollar sign character (**$**) in identifiers. This is no longer permitted; code containing dollar signs in identifiers will not compile correctly under Version 3.0.

2.4.3 Multicharacter **char** constants are allowed under Versions 2.03 and earlier, but are not permitted under Version 3.0. Code containing multicharacter **char** constants will not compile correctly under Version 3.0.

2.5 Every C string is unique. A string can initialize an array and can be modified at run time. Version 3.0 gives every string separate storage, whether or not a string is identical to another string in the program. Versions 2.03 and earlier detect whether two strings are the same and store only one instance of the string. Existing programs that depend on common storage for identical string literals will not run properly under Version 3.0.

4 Version 3.0 implements the **char** type as a signed quantity, and provides the **unsigned char** type to represent unsigned quantities of the same size. Versions 2.03 and earlier treat the **char** type as unsigned. Programs that depend on the **char** type being unsigned will not run properly under Version 3.0.

Version 3.0 implements the **unsigned long** type, a feature not provided in previous versions.

The enumeration type is also provided in Version 3.0. Previous versions did not

offer this feature.

7.1 Under Version 3.0, an array or function identifier is considered an address: a constant pointer to the named array or procedure. You can express the address of the array or function simply by giving the identifier. Under Versions 2.03 and earlier, the address-of operator (&) must be applied to an array or function name to express the address of the array or function. Expressions that use this convention will produce unexpected results under Version 3.0.

In Version 3.0, the name of a structure or union variable represents the *value* of that structure or union. In versions 2.03 and earlier, the name of a structure or union represents the *address* of the structure or union. Expressions that depend on this convention will produce unexpected results under Version 3.0.

7.2 In Version 3.0, casting a value to a pointer type produces an lvalue. This was not true in previous versions.

7.6 – 7.7 The relational and equality operators perform the usual arithmetic conversions in Version 3.0. In Versions 2.03 and earlier, the right operand is converted to the type of the left operand.

8.5 Version 3.0 allocates bitfields low order to high order, whereas versions 2.03 and earlier allocate bitfields high order to low order.

11.2 Version 3.0 differs from earlier versions in its treatment of uninitialized variables declared outside of functions (at the external level). A variable declaration at the external level that lacks both a storage class specifier and initializer is treated either as a reference to a definition of the variable elsewhere in the program or, if no definition appears, as a

"communal" variable that is allocated storage by the linker and initialized to zero when the program is loaded. In previous versions, the variable was allocated storage and initialized at compile time.

12

Version 3.0 adds several new features to the C preprocessor. The special constant-expression "**defined**(*identifier*)" can follow any **#if** or **#elif** directive. The line "#if defined(ANYTHING)" has the same effect as "#ifdef ANYTHING".

The new directive **#elif** directive allows for "else-if" clauses in **#if** blocks.

In Version 3.0, the pound sign (#) introducing the preprocessor directive can be preceded on the same line by any combination of tabs and spaces. In previous versions, the pound sign had to be the first character of the line.

Macro definitions can occupy more than one line in Version 3.0. The newline character is preceded by a backslash (\) to indicate continuation. Earlier versions do not allow continuation.

14.1

Under Version 3.0, structures and unions can be assigned values, passed as arguments to functions, and returned from functions. Earlier versions do not support these features.

14.3

Version 3.0 allows you to check array limits by comparing pointer values against the address just beyond the end of the array. Earlier versions also allow this, but they warn that you have exceeded the array bounds. For example, the following code fragment checks for the bounds of an array.

```
int a[MAX];
proc()
{
        int *p;
        .
        .
        .
        if (p < &a[MAX]) {
        .
        .
        .
        }
}
```

Version 3.0 accepts this construction, while earlier versions generate a warning message.

15

The logical AND and OR operators (&& and | |, respectively) can be used in constant-expressions. These operators were inadvertently omitted from the language reference in previous versions.

Miscellaneous

Version 3.0 makes conservative assumptions about aliases through pointer variables. This means that, in the optimizing stage, the compiler assumes that a memory location referenced indirectly through a pointer variable may also be referenced directly, by another name. The possibility that a program uses aliases (references to the same location by different names) restricts some of the compiler's optimizing procedures. You can use a command line option with Version 3.0 to override the conservative assumptions, allowing the compiler greater freedom in optimization.

Earlier versions do not make any assumptions about aliases and do not restrict optimization.

D.3 Run-Time Library Differences

Many of the library routines documented in Version 2.03 and earlier will run without change under Microsoft Version 3.0. However, some routines are supported in Version 3.0 under a different function name or syntax. These routines are described in detail below. The changes to the routines are designed to provide greater compatibility with UNIX and XENIX standard libraries.

The include file V2TOV3.H provided with your compiler software allows you to use the modified routines in their original form under Version 3.0. In many cases, you can convert your programs by including V2TOV3.H, without having to change your source code.

Some routines supported under Version 2.03 are not supported at all under Version 3.0. The following routines from Version 2.03 are *not* supported in any form under Version 3.0.

Version 2.03 Routines	Definition
allmem	Level 2 memory allocation
getmem	Level 2 memory allocation
peek	Utility
poke	Utility
rlsmem	Level 2 memory allocation
rbrk	Level 1 memory allocation
repmem	Utility
rstmem	Level 2 memory allocation
sizmem	Level 2 memory allocation
stcarg	String manipulation
stccpy	String manipulation
stcd_i	String manipulation
stch_i	String manipulation
stci_d	String manipulation
stcpam	String manipulation
stcpm	String manipulation
stcu_d	String manipulation
stpblk	String manipulation
stpsym	String manipulation
stptok	String manipulation
stspfp	String manipulation

If your program uses any of the above routines, you must provide your own definition of the routine or alter the code to remove the call to the routine.

The following functions and macros are supported in Version 3.0 in a slightly different manner than in previous versions. The routines are listed under their Version 2.03 names; the sections that follow describe the differences between the versions.

abs	iscsym	movmem	stcis	stpchr
creat	iscsymf	open	stcisn	stscmp
fopen	max	setmem	stclen	
freopen	min	setnbuf	stpbrk	

Use the include file V2TOV3.H, or the appropriate definitions from V2TOV3.H, if your program calls any of the above routines. The remainder of this section summarizes the changes to each of the above routines, and lists the corresponding definition from V2TOV3.H that provides compatibility.

D.3.1 abs

The macro *abs* is defined in the include file V2TOV3.H as follows.

```
#define abs(a,b) (((a) < 0)) ? -(a) : (a))
```

If *abs* is not defined as a macro, it will be interpreted as a call to the standard math library function *abs*.

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.2 creat

The previous version of this function differs from the Version 3.0 in two ways. In Version 3.0, the permission bits specified in the *mode* argument control access to the created file. For example, if a file is opened for reading, an attempt to write to the file causes an error. Versions 2.03 and earlier do not guarantee this interpretation of the permission bits.

Versions 2.03 and earlier allow the user to create a file in binary mode by giving the O_RAW flag in the *creat* call. The flag can be joined with the permission setting arguments with the bitwise OR operator (|).

Version 3.0 maintains a distinction between the permission setting of a file and the file translation mode. You can specify the permission setting of a file when you create it using the *creat* routine, but you cannot join the translation flag with the file permission setting. The *creat* routine creates a file in the current default mode, whether that is text mode or binary mode. You can change the default mode for a single file with the *setmode* function, or change the default mode for all opened files from text mode to binary mode by linking with BINMODE.OBJ. BINMODE.OBJ is discussed in Section 7.10 of Chapter 7, "Advanced Topics."

You can also control the translation mode of a particular file by giving an appropriate flag when you open the file. See the discussion of *open* later in this section.

The O_RAW flag is renamed to O_BINARY in Version 3.0. V2TOV3.H can be included to define O_RAW as O_BINARY. However, the user is responsible for removing the O_RAW flag from calls to *creat* and for providing appropriate calls to *open* instead.

Example

```
int infile;
        /* VERSIONS 2.03 AND EARLIER */
infile = creat("test.dat", O_RAW);

        /* EQUIVALENT CALL IN VERSION 3.0 */
infile = open ("test.dat",
          (O_CREAT | O_TRUNC | O_BINARY | O_RDWR),
          (S_IREAD | S_IWRITE));
```

This example shows a call to *creat* in Version 2.03 and an equivalent call in Version 3.0. The call to *open* specifies the O_CREAT and O_TRUNC flags, thus accomplishing the same task as the *creat* call. Using *open* rather than *creat* is recommended for new code.

D.3.3 fopen, freopen

In Version 3.0, when a file is opened for appending ("a" or "at" type), all write operations take place at the end of the file. Although the file pointer can be repositioned using *fseek* or *rewind*, the file pointer is always moved back to the end of the file before any write operation is carried out.

In Version 2.03 and earlier, when a file is opened for appending, the file pointer is initially positioned at the end of the file. All write operations take place at the current position of the file pointer; if the file pointer is repositioned (using *fseek* or *rewind*), any write operations will be carried out at the new position.

D.3.4 iscsym, iscsymf

These macros are extensions to the character classification (*ctype*) macros. Version 3.0 does not include the *iscsym* and *iscsymf* macros in the *ctype* set, but you can continue to use them by including the file V2TOV3.H along with the CTYPE.H file.

The V2TOV3.H file defines these macros as follows.

```
#define iscsym(c)       (isalnum(c) || ((c) == '_'))
#define iscsymf(c)      (isalpha(c) || ((c) == '_'))
```

The Version 3.0 definition of *iscsymf* produces a slightly different result than produced by previous versions since Version 3.0 does not allow the dollar sign ($) as a character in identifiers. Note that if the argument *c* has side effects the results of these macros are unpredictable.

D.3.5 max

The macro *max* is defined in the include file V2TOV3.H as follows.

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.6 min

The macro *min* is defined in the include file V2TOV3.H as follows.

```
#define min(a,b) (((a) < (b)) ? (a) : (b))
```

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.7 movmem

This routine copies a specified number of characters from a given source string to a given destination string. The *movmem* routine handles the transfer correctly in cases where the source and destination strings overlap.

The Version 3.0 routine *memcpy* performs the same task as the *movmem* routine, but the arguments are given in a different order. The include file V2TOV3.H defines *movmem* as follows.

```
#define movmem(s, d, n)          memcpy(d, s, n)
```

D.3.8 open

The *open* routine has the same basic form and function in Version 3.0 as it does in earlier versions, with two exceptions.

1. The flag for binary mode is named O_BINARY instead of O_RAW.
2. The *pmode* argument is required when O_CREAT is specified.

To process a program that uses the O_RAW flag in the call to *open*, include V2TOV3.H or the following definition in your program.

```
#define O_RAW    O_BINARY
```

The Version 3.0 *open* routine takes a third argument. The third argument gives the permission setting of the file; it is optional except when using the O_CREAT flag to create a new file.

D.3.9 setmem

This routine sets a specified number of bytes in a buffer to a given character. The Version 3.0 routine *memset* performs the same task as the *setmem* routine, but the arguments are given in a different order. The include file V2TOV3.H defines *setmem* as follows.

```
#define setmem(p, n, c)          memset(p, c, n)
```

D.3.10 setnbuf

The *setnbuf* routine sets up an empty buffer, and is equivalent to the call

```
setbuf(stream, NULL);
```

Version 3.0 supports the *setnbuf* routine through the following definition in V2TOV3.H.

```
#define setnbuf(stream)  setbuf(stream, NULL)
```

D.3.11 stcis, stcisn, stclen, stpbrk, stpchr, stscmp

These routines are renamed in Version 3.0, but otherwise function exactly the same as in Versions 2.03 and earlier. The names in Version 3.0 are as follows.

Version 2.03 Name	Version 3.0 Name
stcis	strspn
stcisn	strcspn
stclen	strlen
stpbrk	strpbrk
stpchr	strchr
stscmp	strcmp

You can continue using these routines under their Version 2.03 names by including V2TOV3.H or the following definitions in your program.

```
#define stcis(s1, s2)       strspn(s1, s2)
#define stcisn(s1, s2)      strcspn(s1, s2)
#define stclen(s)           strlen(s)
#define stpbrk(s, b)        strpbrk(s, b)
#define stpchr(s, c)        strchr(s, c)
#define stscmp(s1, s2)      strcmp(s1, s2)
```

D.4 Differences in Assembly Language Interface

This section covers the basics of converting assembly language routines written for Versions 2.03 and earlier to run with Version 3.0. Much of the information in this section is also presented in Section 8.1 of Chapter 8, "Interfaces with Other Languages"; this discussion attempts to consolidate the information to make the task of conversion easier. For additional assembly language information not found below, see Section 8.1 of Chapter 8, "Interfaces with Other Languages."

Assembly language routines that are compatible with Versions 2.03 and earlier differ from Version 3.0-compatible routines in five basic areas:

1. Register usage conventions
2. Local variable access (stack setup)
3. Subroutine entry/exit code
4. Global variable naming conventions
5. Segment usage and naming

Each of these areas is discussed in detail below.

D.4.1 Register Usage Conventions

The S and P model programs of Version 2.03 correspond to the small and medium model programs of Version 3.0. In S and P model programs under Version 2.03, ES is always assumed to point to the same segment as SS and DS. However, the "mixed model" programming supported by Version 3.0 allows data in segments outside DS to be accessed. In mixed model programs, the compiler uses ES to reference data outside of the data segment (DS). Thus, ES may not always contain the same value as SS and DS. (Note that SS and DS always contain the same value in Version 3.0 small and medium model programs, unless specifically overridden with the "u" or "w" letter in the /A option.)

Version 3.0 also expects the direction flag of the 8086/8088 processor to be cleared at all times. Therefore, if the assembly routine sets the direction flag, it must clear it (using the CLD instruction) before calling or returning to a C function. This is not required in Version 2.03.

Version 3.0 implements register variables, which were not available in previous versions. The Version 3.0 compiler allows up to two register variables per function. (More than two may be declared, but the extra register requests will be ignored.)

The compiler uses the SI and DI registers to store any register variables. This means that any routine that uses either the SI or DI register must save the register contents upon entry to the subroutine and must restore the original contents before exiting. The compiler takes care of this automatically for C routines, but the user is responsible for providing the necessary instructions in assembly routines. Any assembly routine called from a C function that uses either or both of the SI and DI registers should push the values of the registers onto the local stack (after the stack

is set up) and pop them off the stack before returning to the calling routine.

In the reverse case, where an assembly routine calls a C function, these instructions are not necessary, since the C function automatically saves and restores SI and DI. Since the assembly routine can rely on the values in these registers being preserved across calls to C routines, the registers may not need to be reloaded as often. This assumption may allow more efficient register usage in the assembly routine. See Section D.4.2 for an example.

D.4.2 Stack Setup and Subroutine Entry/Exit Code

All versions of the C compiler use BP as a "frame pointer." Local variables and parameters (also called the "stack frame") are always accessed using offsets from the BP register. However, Version 3.0 differs from earlier versions of the compiler in the entry/exit sequences for subroutines and in the setup of the local stack for subroutine calls.

Figures D.1 and D.2 show the stack frame setups under Version 2.03 and Version 3.0, respectively.

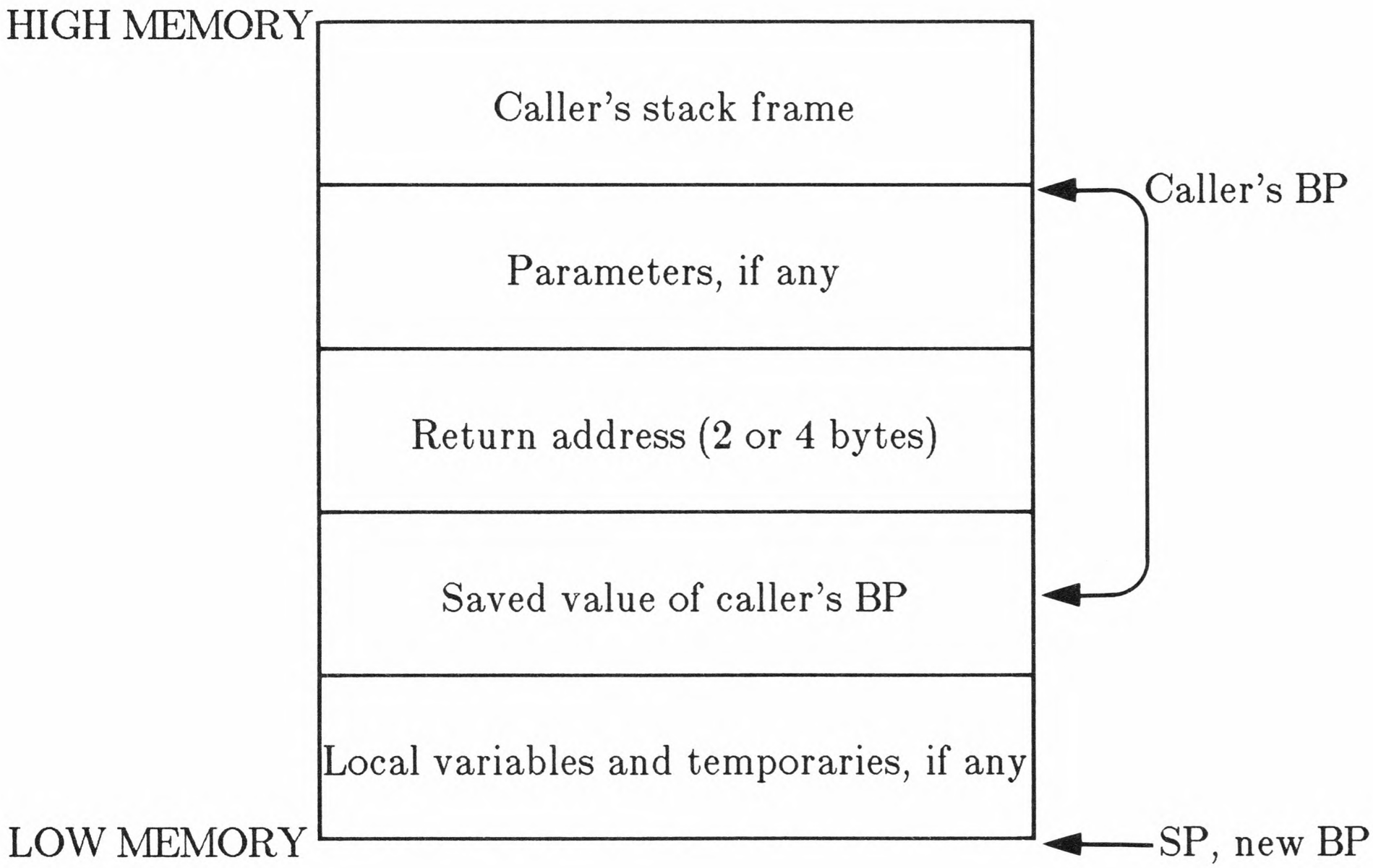

Figure D.1 Version 2.03 Stack Frame Setup

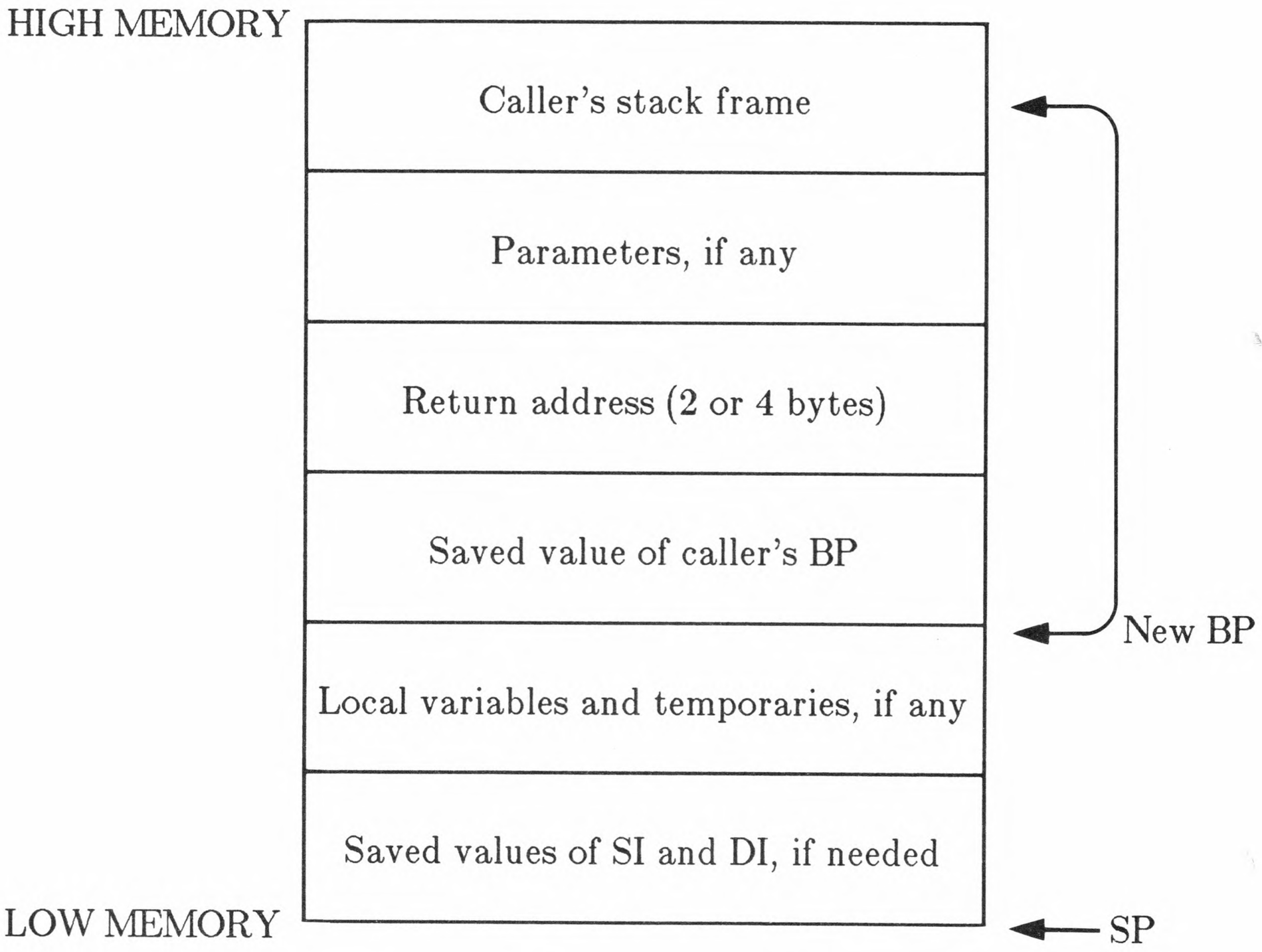

Figure D.2 Version 3.0 Stack Frame Setup

The differences in these two setups are reflected

1. in the entry and exit code sequences for subroutine calls; and
2. in the locations for local variables and parameters.

In Version 3.0, parameter references are positive offsets from BP. Local variable references are negative offsets from BP. The first parameter occurs at either [BP+4] or [BP+6], depending on whether the routine was called using a near call (2-byte address) or a far call (4-byte address).

In Version 2.03, all parameters and local variables are referenced via positive offsets from the BP register. The offset to the first parameter can be calculated as $(n + 4)$ or $(n + 6)$, depending on whether the routine is accessed by a near or far call. The value n is the number of bytes of local storage allocated following the saved caller's frame pointer. Frequently n is zero, in which case parameter offsets in Versions 2.03 and 3.0 are identical.

The versions also differ in the handling of stack checking and stack allocation for local variables other than parameters. In Version 2.03, when stack overflow checking is enabled, the number of bytes of local storage desired (which should be a positive even number in all cases) is subtracted from the stack pointer (SP). The resulting value is compared to a predefined limit value. If the value is less than the limit, a routine named XCOVF is called to report the stack overflow error and terminate the program. If stack checking is disabled, the number of bytes is subtracted from the stack pointer and no overflow checking is performed.

Version 3.0 uses the *__chkstk* routine for stack checking. (The *__chkstk* routine was chosen to help ensure compatibility with XENIX C compilers.) The *__chkstk* routine performs stack checking and produces an error message when appropriate. If stack overflow checking is enabled (the default), the number of bytes of stack space desired is stored in the AX register and the *__chkstk* routine is called. The *__chkstk* routine determines if the request will cause the stack to overflow. If so, *__chkstk* produces an error message to this effect and terminates the program. Otherwise the routine subtracts the given value from the stack pointer and returns. If stack checking is disabled (using the /Gs or /Ox option), the compiler simply subtracts the requested number of bytes from the stack pointer and continues.

Because of the differences in stack setups, exit sequences for Version 3.0 also differ from previous versions. In Version 3.0, the called routine sets SP to the same value as BP. This has the effect of removing local variables from the stack and causing SP to point to the location where the caller's BP was stored. The called routine then pops the caller's saved frame pointer back into BP and returns. The calling routine is responsible for readjusting SP by adding the number of bytes of arguments that were pushed.

In Version 2.03 the called routine first adds the number of bytes of local variables and temporaries to SP, thus causing SP to point to the location of the saved caller's frame pointer. Then the called routine pops the saved frame pointer into BP and returns. After the return, the calling routine must restore the stack pointer by copying the value of BP into SP.

The examples below show typical entry/exit sequences for Versions 2.03 and 3.0. Both examples assume that stack checking is disabled and that 8 bytes is the amount of local variable space required.

Version 2.03

```
ENTRY   push bp         ;save caller's frame pointer (BP)
        sub sp,8        ;allocate local variable space on stack
        mov  bp,sp      ;new frame pointer points to bottom
                        ;  of stack
          .
          .
          .
EXIT    add sp,8        ;deallocate local variable space
        pop  bp         ;restore caller's frame pointer
        ret             ;appropriate to type of call
```

Version 3.0

```
ENTRY   push bp         ;save caller's frame pointer (BP)
        mov  bp,sp      ;frame pointer points to old BP
        sub  sp,8       ;allocate local variable space on stack
        push di         ;required only if routine changes di
        push si         ;required only if routine changes si
          .
          .
          .
EXIT    pop  si         ;required only if si saved on entry
        pop  di         ;required only if di saved on entry

        mov  sp,bp      ;remove local variable space
        pop  bp         ;restore caller's frame pointer
        ret             ;appropriate to type of call
```

Despite the differences listed above, it is not strictly necessary to change the entry/exit sequence of your assembly routines from Version 2.03 to Version 3.0 *unless* your routines attempt to check for stack overflow or use the SI and DI registers. For all other contexts, the setup of the local stack is irrelevant. The parameters are pushed onto the stack in the same way in both versions; the local variable access method is always defined by the routine itself, so any method can be used. The exit sequences of both versions work in essentially the same manner and return to the calling routine with the stack pointer in the same position.

However, changing your code to conform to the Version 3.0 format is still recommended. Debugging will be much easier if your programs consistently use one stack format instead of two.

D.4.3 Global Variable Naming Conventions

In Version 2.03 and earlier, a global name such as XYZ causes a public definition of the name XYZ to be put in the object module. In Version 3.0, for reasons of compatibility with XENIX compilers, an underscore is added to the beginning of the global name when the public definition is put in the object module. For example, the global name XYZ in the source file produces a public definition for the name _XYZ in the object module.

The underscore convention in Version 3.0 means that the name of any assembly routine called from a Version 3.0 program must be defined with a leading underscore in the assembly routine. The C program calls the assembly routine *without* the leading underscore, since the underscore is automatically added by the compiler. For example, the name of an assembly routine might be defined as *_strdo*; the corresponding call in the C program would be "strdo (...)".

Another difference between the compilers arises in the area of case sensitivity. In Version 2.03, external names are not case-sensitive; in Version 3.0, they are. However, when invoking the linker directly (through the LINK command) with a Version 3.0 program, case is ignored by default. You can take advantage of this behavior when linking programs from Version 2.03 and earlier. By contrast, the Version 3.0 compiler control program CL.EXE, which can be used to invoke the linker, automatically tells the linker *not* to ignore case.

Some assemblers are not sensitive to the case of external names, so care must be taken when defining the name of an assembly routine in a C source program.

D.4.4 Segment Usage and Naming

The structure of a Version 3.0 program in memory differs slightly from the Version 2.03 structure. Version 3.0 has the same structure in all memory models. In Version 2.03 there are two different memory layouts, depending on the memory model used. The S and P models in Version 2.03, which correspond to the small and medium models in Version 3.0, use a different layout from Version 3.0. The Version 2.03 D and L models use essentially the same layout as do Version 3.0 programs, with two exceptions:

1. There is no equivalent to the Version 3.0 segment for far data.
2. In Version 2.03 and earlier, SS always points to the stack segment base instead of DS. This is similar to specifying the letter "w" with the memory model (/A) option in Version 3.0.

The Version 2.03 S and P model layout and the Version 3.0 layout are shown in Figures D.3 and D.4, respectively.

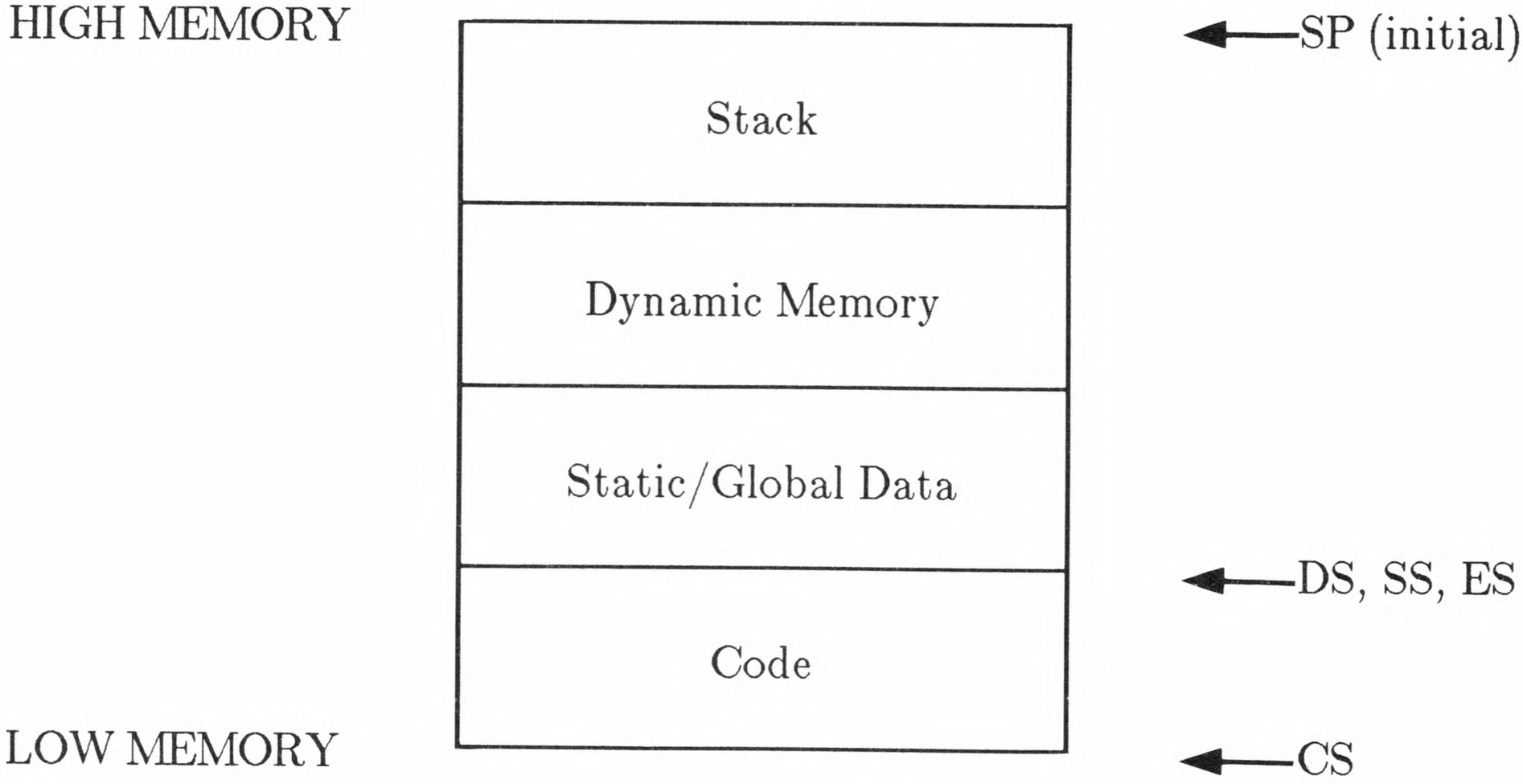

Figure D.3 Version 2.03 S and P Model Layout

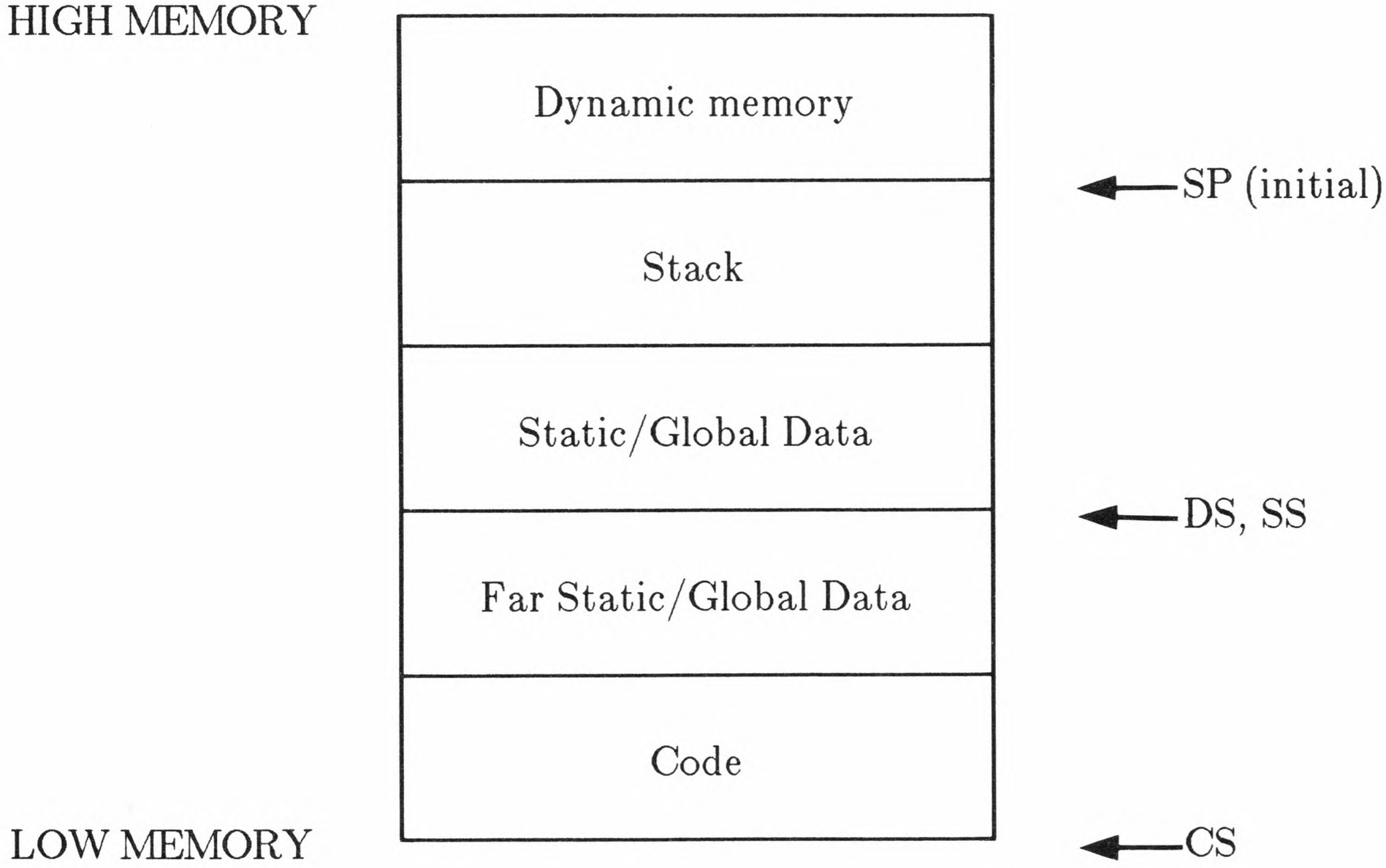

Figure D.4 Version 3.0 Layout

There are two main differences between the above layouts. First, in Version 2.03, the stack resides above dynamic memory. In Version 3.0 it resides below dynamic memory. The Version 3.0 layout means that a program that uses little or no dynamic allocation requires much less space for execution.

The second difference is more important for assembly programmers. In Version 3.0, ES does not necessarily contain the same value as DS. Version 3.0 supports the concept of "far" data in small and medium model programs, while Version 2.03 does not. When far data items are referenced, ES is used to hold the segment value for the far item. Since the compiler has no way of knowing in advance that no far data will occur in the program, it does not rely on ES being the same as DS. Instead, the compiler loads ES whenever it is needed.

Assembly routines written to run with Version 2.03 S and P model programs may have relied on ES being the same as DS. Under Version 3.0, they must load DS into ES to be safe.

Some additional differences between the compilers in the naming of segments and classes are as follows.

- In Version 2.03 the code segments in an S model program are all given class PROG. In Version 3.0 all code segments have class CODE. In Version 3.0 small model, the code segments are all named _TEXT by default; in medium and large model, each compiland forms a segment named *module-name*_TEXT.
- In Version 2.03 S model programs, the code segment is grouped into a group named PGROUP. In Version 3.0 small model programs, the code segment is not grouped.

Both versions use DGROUP to group the DATA and STACK segments in all models.

The general rules and methods for accessing segments in both versions are the same. Usually, the programmer should only be accessing the CODE, _DATA, BSS, c_common, and STACK segments. (Other data segments with class FAR_DATA or FAR_BSS can be useful in some cases.) See Section 8.1.1, "Segment Model," in Chapter 8, "Interfaces with Other Languages," for more information on what kinds of data items are stored in each of the segments.

Appendix E
Error Messages

E.1 Introduction 225
E.2 Run-Time Error Messages 225
E.2.1 Run-Time Library Error Messages 226
E.2.2 Floating-Point Exceptions 227
E.2.3 Run-Time Limits 229
E.3 Compiler Error Messages 230
E.3.1 Warning Error Messages 232
E.3.2 Fatal Error Messages 239
E.3.3 Compilation Error Messages 241
E.3.4 Command Line Messages 251
E.3.5 Compiler Limits 253
E.4 Linker Error Messages 255
E.5 Library Manager Error Messages 261
E.6 EXEPACK Error Messages 263
E.7 EXEMOD Error Messages 264

E.1 Introduction

This appendix lists error messages you may encounter as you develop a program and gives a brief description of the action to take to correct the error. The first section lists run-time errors. Run-time errors are errors you encounter when you execute your program.

The remaining sections describe errors generated by the following programs:

- The Microsoft C Compiler
- The Microsoft LINK utility
- The Microsoft LIB utility
- The EXEPACK utility
- The EXEMOD utility

E.2 Run-Time Error Messages

Run-time error messages fall into four categories:

1. Error messages generated by the run-time library to notify you of serious errors. These messages are listed and described below.
2. Floating-point exceptions generated by the 8087/80287 hardware or the emulator. These exceptions are listed and described in Section E.2.2.
3. Error messages generated by calls in the program to error-handling routines in the C run-time library (the *abort, assert,* and *perror* routines.) These routines print an error message to standard error whenever the program calls the given routine. For a description of these routines and the corresponding error messages, see the *Microsoft C Run-Time Library Reference.*
4. Error messages generated by calls to math routines in the C run-time library. On error, the math routines return an error value and some print a message to the standard error. See the *Microsoft C Run-Time Library Reference* for a description of the math routines and corresponding error messages.

E.2.1 Run-Time Library Error Messages

The following messages may be generated at run time when your program has serious errors.

Floating point not loaded

Your program needs the floating-point library, but the library was not loaded. The error causes the program to be terminated with an exit status of 255. This occurs in two situations:

1. A format string for one of the routines in the *printf* or *scanf* family contains a floating-point format specification and there are no floating-point values or variables in the program. The C compiler attempts to minimize the size of a program by loading floating-point support only when necessary. Floating-point format specifications within format strings are not detected, so the necessary floating-point routines are not loaded. To correct this error use a floating-point argument to correspond to the floating-point format specification. This causes floating-point support to be loaded.

2. *X*LIBFP.LIB or *X*LIBFA.LIB (where *X* is S, M, or L, depending on the memory model) was specified after *X*LIBC.LIB in the linking stage. You must relink the program with the correct library specification.

Null pointer assignment in program

The contents of the NULL segment have changed in the course of program execution. The NULL segment is a special location in low memory that is not normally used. If the contents of the NULL segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Notice that your program can contain null pointers without generating this message; the message appears only when you access a memory location through the null pointer.

This error does not cause your program to terminate; the error message is printed following the normal termination of the program.

This message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

Stack overflow
: Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program is terminated with an exit status of 255. To correct the problem, relink using the linker /STACK option to allocate a large stack, or modify the stack information in the executable file header by using the EXEMOD program.

E.2.2 Floating-Point Exceptions

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions.

Using C's default 8087/80287 control word settings, the following exceptions are masked and do not occur.

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Precision	Exception masked

The following errors do not occur with code generated by the Microsoft C Compiler or provided in the Microsoft C Run-Time Library.

Square root

Stack underflow

Unemulated

The floating-point exceptions are listed and described below.

Floating point error: Denormal
: A very small floating-point number was generated, which may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on.

Floating point error: Divide by 0
: An attempt was made to divide by zero.

Floating point error: Integer overflow
Overflow on assigning a floating-point value to an integer.

Floating point error: Invalid
Invalid operation; usually involves operating on NaN's or infinities.

Floating point error: Overflow
Overflow in floating-point operation.

Floating point error: Precision
Loss of precision occurred in a floating-point operation. This exception is normally masked, since almost any floating-point operation can cause loss of precision.

Floating point error: Stack overflow
A floating-point expression has used too many stack levels on the 8087/80287 or emulator. (Stack overflow exceptions are trapped up to a limit of seven additional levels beyond the eight levels normally supported by the 8087/80287 processor.)

Floating point error: Stack underflow
A floating-point operation resulted in a stack underflow on the 8087/80287 or emulator.

Floating point error: Square root
The operand in a square root operation was negative. (Note: the *sqrt* function in the C run-time library checks the argument before performing the operation and returns an error value if the operand is negative; see the *Microsoft C Run-Time Library Reference* for details on *sqrt.)*

Floating point error: Underflow
Underflow in a floating-point operation. (An underflow is normally masked so that the operation yields the result 0.0.)

Floating point error: Unemulated
An attempt was made to execute an 8087/80287 instruction not supported by the emulator or an invalid 8087/80287 instruction.

E.2.3 Run-Time Limits

Table E.1 summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.1

Program Limits at Run Time

Program Item	Description	Limit
Files	Maximum file size	$2^{32}-1$ bytes (4 gigabytes)
	Maximum number of open files (streams)	20[a]
Command Line	Maximum number of characters (including program name)	128
Environment Table	Maximum size	32K

[a] Five streams are opened automatically (*stdin, stdout, stderr, stdaux,* and *stdprn*), leaving 15 available for the program to open.

E.3 Compiler Error Messages

The error messages produced by the C compiler fall into five categories:

1. Warning messages
2. Fatal error messages
3. Compilation error messages
4. Command line error messages
5. Compiler internal error messages

Warning messages are informational only; they do not prevent compilation and linking. You can control the level of warnings generated by the compiler by using the /W option, described in Section 3.9.2 of Chapter 3, "Compiling." The list of warning messages below include a number for each message indicating the minimum level that must be set for the message to appear.

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program. After printing out a message about the fatal error, the compiler terminates without producing an object file or checking for further errors.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. When the compiler encounters a nonfatal program error, it attempts to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler terminates processing.

Command line messages give you information about invalid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler terminates processing.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. The following messages are compiler internal error messages. No matter what your source program contains, these messages should not appear. If they do, please report the condition to Microsoft Corporation, using the Software Problem Report at the back of this manual. Although these errors are not the fault of your program,

you will probably want to rearrange your code so that the program can be compiled.

Compiler error (assertion): file *filename*, line *n* source=*filename*
: The compiler performs internal consistency checks during the course of compilation. This message indicates that the consistency check failed. In this message, the first *filename* identifies the compiler file producing the error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

Compiler error (code generation)
: The compiler could not generate code for an expression. Usually this occurs with a complex expression; try rearranging the expression.

Compiler error (internal)
: The compiler's internal consistency check failed, and the compiler cannot continue.

Fatal(assertion count exceeds 5; stopping compilation)
: When this message appears, more than five assertion errors have accumulated, and the compiler cannot continue operation.

Error messages in the warning, fatal, and compilation error message categories have the same basic form

filename (*linenumber*) : *message*

where *filename* is the name of the source file being compiled, *linenumber* identifies the line of the file containing the error, and *message* is a self-explanatory description of the error or warning. For warning messages and fatal errors, the word "warning" or "fatal" appears at the beginning of the message, followed by a colon.

Command line error messages simply give a message about the command line, so they do not contain references to line numbers and filenames.

The messages for each category are listed below in alphabetical order, along with a brief explanation of each error. To look up an error message, first determine the message category. If the message pertains to the command line, look in the command line section. If the message begins with the word "warning" or "fatal," look in the corresponding section. Otherwise, the message is a compilation error message, and you can find it in that section.

Within each section the error messages are listed alphabetically. Disregard opening single quotes (') when looking up error messages alphabetically.

An error message may begin with an operator or with an item from your program, such as an identifier. In these cases look up the error message under the first complete word following the symbol or program item in the message. For example, to find the description of the message

```
warning : '=' : illegal pointer combination
```

look in the "warning" messages section under "illegal".

If the message begins with a keyword or preprocessor directive, such as **typedef** or **#define**, look up the message under the keyword or the first letter of the directive. For example, look up the message

```
#include expected a file name
```

under "include" in the error message section.

Section E.3.5, "Compiler Limits," summarizes limits imposed by the Microsoft C Compiler (for example, the maximum size of a macro definition).

E.3.1 Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear.

warning : address of frame variable taken, DS != SS [1]
: You must use a **far** pointer when taking the address of a frame variable in a program with separate data and stack segments.

warning : array's declared subscripts differ [1]
: An array is declared twice with differing sizes. The larger size is used.

warning : at least one void operand [1]
: An expression with type **void** is used as an operand.

warning : bad storage class *specifier* on function *identifier* [1]
Functions must have **static** or **extern** class; any other storage class specifier is ignored.

warning : bitfield type must be integral [1]
Bitfields must be declared as **unsigned** integral types. A conversion has been supplied.

warning : bitfield type must be unsigned [1]
Bitfields must be declared as **unsigned** integral types. A conversion has been supplied.

warning : cast of int expression to far pointer [1]
A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer produces an address with a meaningless segment value.

warning : constant too big [1]
Information is lost because a constant value is too large to be represented in the type to which it is assigned.

warning : conversion lost segment [1]
The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.

warning : converting a long address to a short address [1]
The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) results in the loss of the segment address.

warning : data conversion [3]
Two data items in an expression had different types, causing the type of one item to be converted.

warning : declared parameter list differs from definition [1]
The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.

warning : different enum types [1]
Two different **enum** types are used in an expression.

warning :'*operator*': different levels of indirection [1]
An expression involving the specified operator has inconsistent levels of indirection. For example,

```
char **p;
char *q;
   .
   .
   .
p=q;  /* different levels of indirection */
```

warning : different types : parameter *n*
The type of the given parameter in a function call does not agree with the argument type list or the function definition.

warning : first parameter list is longer than the second [1]
A function is declared more than once and the argument type lists in the declarations differ.

warning : *identifier* : formal parameter has bad storage class [1]
Formal parameters must have **auto** or **register** storage class.

warning : formal parameter *identifier* is redefined [1]
The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function.

warning : '*identifier*' : formal parameters ignored [1]
Formal parameters appeared in a function declaration (for example, "extern int *f(a,b,c);"). The formal parameters are ignored.

warning : '*identifier*' : function used as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning : function declaration specified variable args [1]
The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.

warning : function must return a value [2]
A function is expected to return a value unless it is declared as **void**.

warning : function *identifier* too large for post-optimizer [0]
The named function was not optimized because insufficient space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : function was declared with formal arguments [1]
The function was declared to take arguments, but the function definition does not declare formal parameters.

warning : function was declared without formal arguments [1]
The function was declared to take no argument (the argument type list consists of the word **void**) but formal parameters are declared in the function definition or arguments are given in a call to the function.

warning : '*identifier*' : has bad storage class [1]
The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context is used in place of the illegal class.

warning : identifier truncated to "*identifier*" [1]
Only the first 31 characters of an identifier are significant.

warning : illegal null char [1]
The single quotes delimiting a character constant must contain one character. For example, the declaration "char a = ' '" is illegal. To represent a null character constant, use an escape sequence (for example, '\0').

warning : '*operator*' : illegal pointer combination [1]
A pointer to a given type is forced to point to an object with a different type.

warning : '*operator*' : illegal with enums [1]
You may not use the given operator with an **enum** value. The **enum** value is converted to **int** type.

warning : *operator* : indirection to different types [1]
The indirection operator (*) is used in an expression to access values of different types.

warning : long/short mismatch in arguments : conversion supplied [1]
An integral type is assigned to an integer of a different size, causing a conversion to take place. For example, a **long** is given where a **short** was declared, etc.

warning : macro *identifier* requires parameters [1]
The given *identifier* was defined as a macro taking one or more arguments, but the *identifier* is used in the program without arguments.

warning : missing close parenthesis after 'defined' [1]
The closing parenthesis is missing from an **#if defined** phrase.

warning : near/far on *identifier* ignored [1]
The **near** or **far** keyword has no effect in the declaration of the given *identifier* and is ignored.

warning : near/far mismatch in arguments: conversion supplied [1]
A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.

warning : newline in string constant [1]
A newline character is not preceded by an escape character (\) in a string constant.

warning : no function return type [2]
The return type is missing from a function declaration; the default return type will be **int**.

warning : no return value [2]
A function declared to return a value does not do so.

warning : not enough actual parameters for macro *identifier* [1]
The number of actual arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning : not enough arguments [1]
The number of arguments specified in a function call is less than the number of parameters specified in the argument type list or in the function definition.

warning : '&' on function/array, ignored [1]
You cannot apply the address-of operator to a function or array identifier.

warning : overflow in constant arithmetic [1]
The result of an operation exceeds 0x7FFFFFFF.

warning : overflow in constant multiplication [1]
The result of an operation exceeds 0x7FFFFFFF.

warning : '*identifier*' : overflows array bounds [1]
Too many initializers are present for the given array. The excess initializers are ignored.

warning : parameter *n* differs
The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.

warning : parameter *n*'s type is not in union *identifier*
The declaration of the given parameter specifies a **union** type, but the parameter's type does not correspond to the type of any of the union members.

warning : pointer mismatch: parameter *n* [1]
The given parameter has a different pointer type than is specified in the argument type list or the function definition.

warning : procedure too large, skipping loop inversion optimization and continuing [0]
Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : procedure too large, skipping branch sequence optimization and continuing [0]
Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : procedure too large, skipping cross jump optimization and continuing [0]
Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : recoverable heap overflow in post optimizer - some optimizations may be missed [0]
Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : *identifier* : redefinition [1]
The given *identifier* is redefined.

warning : 'register' on '*identifier*' ignored [1]
Only integral and pointer type variables may be given **register** storage class.

warning : second parameter list is longer than the first [1]
A function is declared more than once, and the argument type lists in the declarations differ.

warning: sizeof returns 0 [2]
The **sizeof** operator is applied to an operand that yields a size of zero.

warning : string too big, leading chars truncated [1]
A string exceeds the compiler limit on string size. To correct this problem, you must break the string down into two or more strings.

warning : strong type mis-match [2]
Two different but compatible types are used: for example, a **typedef** type with a non-**typedef** type, or two different but equivalent **struct** or **union** types.

warning : too many actual parameters for macro *identifier* [1]
The number of actual arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning : too many actual parameters [1]
The number of arguments specified in a function call is greater than the number of parameters specified in the argument type list or in the function definition.

warning : type following '*keyword*' is illegal, ignored [1]
An illegal combination occurs (for example, **unsigned float**).

warning : #undef expected an identifier
The name of the identifier whose definition is to be removed must be given with the **#undef** directive.

warning : unexpected formal parameter list [1]
A formal parameter list is given in a function declaration and is ignored.

warning : '*identifier*' : unknown size [1]
The size of the named variable is not specified.

warning : unmatched close comment '*/' [1]
A comment is started (with '/*') but is not closed (with '*/').

warning : unnamed struct/union as parameter [1]
The structure or union type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.

warning : *identifier* uses undefined struct/union *identifier* [2]
The name of a structure or union type is used before the type is defined.

warning : '*identifier*' : void type changed to int [1]
Only functions may be declared to have `void` type.

E.3.2 Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

fatal : bad parenthesis nesting
The parentheses in a preprocessor directive are not matched.

fatal : bad preprocessor command '*string*'.
The characters following the number sign (#) do not form a preprocessor directive.

fatal : cannot find *filename*
The given file does not exist or cannot be found. Check to make sure your environment settings are valid and that you have given the correct pathname for the file.

fatal : cannot open *filename*
The given file cannot be opened.

fatal : cannot open listing file *filename*
The filename or pathname given for the listing file is not valid.

fatal : cannot open source file *filename*
The filename or pathname given for the source file is not valid.

fatal : compiler limit : macro expansion too big
The expansion of a macro exceeds the available space.

fatal : compiler limit : possibly a recursively defined macro
The expansion of a macro exceeds the available space. Check to see whether the macro is recursively defined, or if the expanded text is too large.

fatal : DGROUP data allocation exceeds 64K
Large model allocation of variables to the default segment exceeds 64K; use the /Gt option to move items into separate segments.

fatal : error count exceeds *n*; stopping compilation
Errors in the program are too numerous or too severe to allow recovery, and the compiler must terminate.

fatal : error on compiler intermediate file
Not enough disk space is available for the compiler to create intermediate files used in the compilation process. The space required is approximately two times the size of the source file. To correct this problem, you must make more space available.

fatal : expected '#endif'
An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

fatal : #if [n] def expected an identifier
You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal : no input file specified
You must give at least one source file as input to the compiler.

fatal : not able to execute compiler pass
One of the compiler executable files cannot be found. Check your environment settings and directory setup.

fatal : parser stack overflow, please simplify your program
Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, try to simplify your program.

fatal : recursively defined macro *identifier*
The given identifier is defined recursively.

fatal : too many include files
Nesting of **#include** directives exceeds the limit of ten levels.

fatal : too many open files, cannot redirect *filename*
The specified redirection cannot be carried out because the system limit on the number of open files has been reached.

fatal : unexpected '#elif'
The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected '#else'
The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected '#endif'
: An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected EOF
: This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file.

E.3.3 Compilation Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

'+' : 2 pointers
: Two pointers may not be added.

'*identifier*' : array inits require curly braces
: The braces ({}) around an array initializer are missing.

array of functions
: Arrays of functions are not allowed.

auto allocation exceeds 32K
: The space allocated for the local variables of a function exceeds the limit of 32 kilobytes.

'*class*' : bad storage class
: The given storage *class* cannot be used in this context.

operator : bad left operand
: The left-hand operand of the given *operator* is an illegal value.

bad octal number '*n*'.
: The character *n* is not a valid octal digit.

operator : bad right operand
: The right-hand operand of the given *operator* is an illegal value.

'*identifier*' : base type with near/far not allowed
: Declarations of structure and union members may not use the **near** and **far** keywords.

identifier : can't init arrays in functions
Arrays can only be initialized at the external level.

cannot init struct/union in functions
Structures and unions can only be initialized at the external level.

case expression not constant
Case expressions must be integral constants.

case expression not integral
Case expressions must be integral constants.

case value '*n*' already used
The case value *n* has already been used in this **switch** statement.

cast has illegal formal parameter list
A formal parameter list is given in a type cast expression.

cast of 'void' term to non-void
The **void** type may not be cast to any other type.

cast to array type is illegal
An object cannot be cast to an array type.

cast to function returning . . . is illegal
An object cannot be cast to a function type.

compiler error
Compiler errors indicate an error on the part of the compiler rather than your program. See Section E.3, "Compiler Error Messages," for instructions on notifying Microsoft about these errors.

Although this message does not indicate errors in your program, you may want to try rearranging your code so that the program will compile. In the error message, the first *filename* identifies the compiler file producing the error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

compiler limit : initializers too deeply nested
The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

compiler limit : struct/union nesting
Nesting of structure and union definitions may not exceed five levels.

constant expression is not integral
The context requires an integral constant expression.

#define syntax
A **#define** directive has a syntax error.

'*identifier*' : definition too big
Macro definitions may not exceed 256 bytes.

'*operator*' : different aggregate types
Pointers to different structure or union types are not allowed with the given *operator*.

divide by 0
The second operand in a division operation (/) evaluates to zero, giving undefined results.

'*identifier*' : enum/struct/union type redefinition
The given *identifier* has already been used for an enumeration, structure, or union tag.

expected '(' to follow '*identifier*'
The context requires parentheses after the function *identifier*.

expected constant expression
The context requires a constant expression.

expected 'defined(id)'
An **#if** directive has a syntax error.

expected exponent value, not '*n*'
The exponent of a floating-point constant is not a valid number.

expected formal parameter list, not a type list
An argument type list appears in a function definition instead of a formal parameter list.

expected preprocessor command, found '*c*'
The character following a number sign (#) is not the first letter of a preprocessor directive.

'*identifier*' : field has indirection
The bitfield is declared as a pointer (*), which is not allowed.

'*identifier*' : field type too small for number of bits
The number of bits specified in the bitfield declaration exceeds the number of bits in the given **unsigned** type.

'*identifier*' : fields only in structs
Only structure types may contain bitfields.

function returns array
A function may not return an array. (It may return a pointer to an array.)

function returns function
A function may not return a function. (It may return a pointer to a function.)

'*identifier*' : functions are illegal members
A function cannot be a member of a structure; use a pointer to a function instead.

'*string*' : ignored
The given text appeared out of context and was ignored.

illegal allocation of *segment-type* segment > 64K
A segment exceeds the limit of 64 kilobytes. The *segment-type* indicates whether the code or data segment exceeds the limit. To correct this problem you must use a larger memory model.

illegal break
A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

illegal case
The **case** keyword may only appear within a **switch** statement.

illegal cast
A type used in a cast operation is not a legal type.

illegal continue
A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

illegal default
The **default** keyword may only appear within a **switch** statement.

illegal escape sequence
The character(s) after the escape character (\) do not form a valid escape sequence.

illegal expression
: An expression is illegal because of a previous error. (The previous error may not have produced an error message.)

'*operator*' : illegal for struct/union
: Structure and union type values are not allowed with the given *operator*.

illegal index, indirection not allowed
: A subscript was applied to an expression that does not evaluate to a pointer.

illegal indirection.
: The indirection operator (*) was applied to a nonpointer value.

illegal initialization
: An initialization is illegal because of a previous error. (The previous error may not have produced an error message.)

'*operator*' : illegal pointer combination
: Pointers that point to different types cannot be used with the given *operator*.

illegal pointer subtraction.
: Only pointers that point to the same type may be subtracted.

illegal sizeof operand
: The operand of a **sizeof** expression must be an identifier or a type name.

#include expected a file name
: An **#include** directive lacks the mandatory filename specification.

identifier : incompatible types
: An expression contains types that are not compatible.

'*identifier*': init of a function
: Functions may not be initialized.

identifier is an undefined struct/union
: The given *identifier* is declared as a structure or union type that has not been defined.

keyword 'enum' illegal
: The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

identifier : label redefined
The given *identifier* appears before more than one statement in the same function.

label '*identifier*' was undefined.
The function does not contain a statement labeled with the given *identifier*.

left of '->*identifier*' must have a struct/union type
The expression before the member selection operator '->' does not point to a structure or union type.

left of '.*identifier*' must have a struct/union type
The expression before the member selection operator '.' does not evaluate to a structure or union type.

left of '->' specifies undefined struct/union '*identifier*'
The expression before the member selection operator '->' points to a structure or union type that is not defined.

left of '.' specifies undefined struct/union '*identifier*'
The expression before the member selection operator '.' has a structure or union type that is not defined.

operator : left operand must be lval.
The left operand of the given *operator* must be an lvalue.

#line expected a line number
A **#line** directive lacks the mandatory line number specification.

'*identifier*' : member of enum redefinition
The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

missing '>'
The closing angle bracket ('>') is missing from an **#include** directive.

missing name following '<'
An **#include** directive lacks the mandatory filename specification.

missing open paren after keyword 'defined'
Parentheses must surround the identifier to be checked in an **#if** directive.

'*identifier*' : missing subscript
To reference an element of an array you must use a subscript.

mod by 0
The second operand in a remainder operation (%) evaluates to zero, giving undefined results.

more than one default
A **switch** statement contains too many **default** labels (only one is allowed).

'*operator*' needs lvalue.
The given *operator* must have an lvalue operand.

negative subscript
A value defining an array size was negative.

newline in constant
A newline character in a character or string constant must be preceded by the backslash escape character (\).

no closing single quote
A newline character in a character constant must be preceded by the backslash escape character (\).

non-address expression
An attempt was made to initialize an item that is not an lvalue.

non-constant offset
An initializer uses a non-constant offset.

non-integer switch expression
Switch expressions must be integral.

non-integral field initializer *identifier*
An attempt is made to initialize a bitfield member of a structure with a non-integral value.

non-integral index
Only integral expressions are allowed in array subscripts.

'*identifier*' : not a function
The given *identifier* was not declared as a function, but an attempt was made to use it as a function.

'*identifier*' : not struct/union member
The given *identifier* is used in a context that requires a structure or union member.

'&' on bit field ignored
Bitfields cannot have their address taken.

'&' on constant
Only variables and functions can have their address taken.

'&' on register variable
Register variables cannot have their address taken.

out of heap space
The compiler has run out of dynamic memory space. This usually means that your program has many symbols and complex expressions. To correct the problem, break down the file into several smaller source files.

out of macro actual parameter space
Arguments to preprocessor macros may not exceed 256 bytes.

parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeds the limit of 32 kilobytes.

parameter has type void
Formal parameters and arguments to functions may not have **void** type.

pointer + non-integer
Only integral values may be added to pointers.

'*operator*' : pointer on left. Needs integral right.
The left operand of the given *operator* is a pointer; the right operand must be an integral value.

'+' : 2 pointers
Two pointers may not be added.

preprocessor command must start as first non-whitespace
Non-whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

'*identifier*' : redefinition
The given *identifier* was defined more than once.

redefinition of formal parameter *identifier*
A formal parameter to a function is redeclared within the function body.

'&' requires lvalue
The address-of operator can only be applied to lvalue expressions.

'.' requires struct/union name
: The expression before the member selection operator '.' is not the name of a structure or union.

'->' requires struct/union pointer
: The expression before the member selection operator '->' is not a pointer to a structure or union.

reuse of macro formal *identifier*
: The parameter list in a macro definition contains two occurrences of the same identifier.

'-' : right operand pointer
: The right-hand operand in a subtraction operation (-) is a pointer, but the left-hand operand is not.

static procedure '*identifier*' not found.
: A forward reference was made to a missing static procedure.

struct/union inits need curly braces
: The braces ({}) around a structure or union initializer are missing.

struct/union member needs to be inside a struct/union
: Structure and union members must be declared within the structure or union.

struct/union member redefinition
: The same identifier was used for more than one structure or union member.

structure/union comparison illegal
: You cannot compare two structures or unions. (You can, however, compare individual members of structure and unions.)

subscript on non-array.
: A subscript was used on a variable that is not an array.

syntax error
: This statement or the preceding statement is not formed correctly.

term does not evaluate to a function
: An attempt is made to call a function through an expression that does not evaluate to a function pointer.

'*n*' : too big for char
: The number *n* is too large to be represented as a character.

too many chars in constant
: A character constant is limited to a single character or escape sequence. (Multicharacter character constants are not supported.)

too many initializers.
: The number of initializers exceeds the number of objects to be initialized.

typedef specifies different enum
: Two different enumeration types defined with **typedef** are used to declare an item, but the enumeration types are different.

typedef specifies different struct
: Two structure types defined with **typedef** are used to declare an item, but the structure types are different.

typedef specifies different union
: Two union types defined with **typedef** are used to declare an item, but the union types are different.

'typedefs' both define indirection
: Two **typedef** types are used to declare an item and both **typedef** types have indirection. For example, the declaration of *p* in the following example is illegal.

```
typedef int *P_INT;
typedef short *P_SHORT;
/* this declaration is illegal */
P_SHORT P_INT p;
```

'*identifier*' : undefined
: The given *identifier* is not defined.

'*c*' : unexpected in formal list
: The character *c* is misused in a macro definition's list of formal parameters.

'*c*' : unexpected in macro definition
: The character *c* is misused in a macro definition.

unknown character '0x*n*'
: The given hexadecimal number does not correspond to a character.

'*identifier*' : unknown size
: A member of a structure or union has an undefined size.

use of undefined struct/union *identifier*
: The given *identifier* was used to refer to a structure or union type that is not defined.

'void' illegal with all types
: The **void** type cannot be used in operations with other types.

'*expression*' was the use of the struct/union
: An undefined structure or union type variable is used in the given *expression*.

E.3.4 Command Line Messages

The following messages indicate errors on the command line used to invoke the compiler. If possible, the compiler continues operation, printing a warning message. In some cases, command line errors are fatal and the compiler terminates processing.

80286 selected over 8086 for code generation
: Both /G1 and /G2 were specified; /G2 is selected.

a previously defined model specification has been overridden
: Two different memory models are specified; the model specified later is used.

argument list for *name* too big
: The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

assembly files are not handled
: You cannot give assembly source files as input to the compiler.

-C ignored (must also specify -P or -E or -EP)
: The -C option takes effect only when you are creating a preprocessed listing using -P, -E, or -EP.

could not execute *name*. Please insert diskette and hit any key.
: One of the compiler passes cannot be found on the current disk. Insert the disk containing the named file and press any key.

ignoring unknown switch *identifier*
: One of the options given on the command line is not recognized and is ignored.

incomplete model specification
: Either one capital letter (S, M, or L) or a string of three lowercase letters must be specified with the /A option.

listing has precedence over assembly output
: Two different listing options were chosen; the assembly listing is not created.

-ND not allowed with -Ad
: The -ND option cannot be used when "d" or "w" is specified with the -A option.

-ND not allowed with -Aw
: The -ND option cannot be used when "d" or "w" is specified with the -A option.

only one floating-point model allowed
: You can only give one of the five floating-point (/FP) options on the command line.

only one of -P/-E/-EP allowed, -P selected
: Each of these options produces a different kind of preprocessed listing; only one can be used at a time.

only one memory model allowed
: You must choose one memory model; you cannot specify more than one.

optimizing for space over time
: This message confirms that the /Os option is used for optimizing.

threshold only for far/huge data, ignored
: The threshold option (/Gt) is effective only in large model programs.

too many linker flags on command line
: Too many linker options are specified with the CL command; not all of them can be passed to the linker.

too many symbols predefined with -D
: The limit on command line definitions is normally 16; the /U or /u option can be used to increase the limit to 20.

unknown -A subswitch '*c*'
: One of the letters given with the -A option is not recognized.

unknown floating-point option
: The specified floating-point option (an /FP option) is not one of the five valid options.

unknown -M substring '*c*'
One of the letters given with the -M option is not recognized. (The -M option is a XENIX-compatible option.)

unknown option (*c*) in *option*
One of the letters in the given option is not recognized.

E.3.5 Compiler Limits

To operate the Microsoft C Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table E.2 summarizes the limits imposed by the C compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.2
Limits Imposed by the C Compiler

Program Item	Description	Limit
String Literals	Maximum length of a string, including the terminating null character (\0).	512 bytes
Constants	Maximum size of a constant is determined by its type; see the *Microsoft C Language Reference* for a discussion of constants.	
Identifiers	Maximum length of an identifier.	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions.	5 levels
Preprocessor Directives	Maximum size of a macro definition.	512 bytes
	Maximum number of actual arguments to a macro definition.	8 arguments
	Maximum length of an actual preprocessor argument.	256 bytes
	Maximum level of nesting for **#if**, **#ifdef**, and **#ifndef** directives.	32 levels
	Maximum level of nesting for include files.	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

E.4 Linker Error Messages

All error messages, except for warning messages, cause the link session to end. After you locate and correct a problem, you must relink.

Messages appear in the map file and are displayed on the screen. If you direct the map file to CON, the error messages will not be displayed on the screen.

About to generate .EXE file
Change diskette in drive A: and press <ENTER>.
: This message appears before the ".EXE" is written if the /P switch is given. Insert the disk the ".EXE" file is to be written to into the specified drive (A: for example).

Ambiguous switch error: *switch*
: You did not enter a unique switch name prefix after the switch indicator /, such as LINK /N ALPHA; linker will abort.

Array element size mismatch
: A far communal array is declared with two or more different array element sizes (e.g., declared once as an array of characters and once as an array of floating-point values). Reconcile the definitions and recreate object module.

Attempt to put segment *name* in more than one group in file *filename*
: A segment is declared to be a member of two different groups. Correct the source and recreate the object files.

Bad value for cparMaxAlloc
: The number specified using the /CPARMAXALLOC switch does not lie in the range 1 to 65,535. Try again.

Cannot find library: *name*.lib
Enter new file spec.
: The linker cannot find the given library and is giving you a chance to specify a new file name or a new path specification or both. You should respond to the prompt with a new file name or a new path specification or both.

Cannot nest response files
: You have named a response file within a response file, which is illegal.

Cannot open list file
: The directory or disk is full. Make space on the disk or in the directory.

Cannot open response file
: You named a response file the linker cannot open. Try again.

Cannot open run file
: The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file
: The directory or disk is full. Make space on the disk or in the directory.

Cannot reopen list file
: You did not actually replace the original disk when prompted. Restart the linker.

Common area longer than 65536 bytes
: Your program has more than 64 kilobytes of communal variables. NOTE: at the present time, only Microsoft C programs can cause this message to be displayed. Rewrite your program using fewer communal variables or making some of your communal variables **far**; or recompile your program large model.

Dup record too large
: LIDATA record contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (e.g. alpha db 10 dup (11 dup (12 dup (13 dup (...))))). Simplify and reassemble.

Fixup overflow near *address* in segment *name* in *file*.OBJ(*file*) offset *offset*
: Some possible causes:

  A group is larger than 64 kilobytes.

  The user's program contains an intersegment short jump or intersegment short call.

  The user has a data item whose name conflicts with that of a subroutine in a library included in the link.

  In an assembly language source file, the user has an EXTRN declaration inside the body of a segment. For example,

```
beta    segment public 'code'
extrn   bar:far
alpha   proc    far
        call    bar
        ret
alpha   endp
beta    ends
```

The following construction is preferable.

```
extrn   bar:far
beta    segment public 'code'
alpha   proc    far
        call    bar
        ret
alpha   endp
beta    ends
```

To correct the problem, revise the source and recreate the object file.

Incorrect DOS version, use DOS 2.0 or later
: Linker will run only on MS-DOS Version 2.0 or later. Reboot your system with MS-DOS 2.0 or later and try linking again.

Insufficient stack space
: There is not enough memory to run the linker.

Interrupt number exceeds 255
: A number greater than 255 has been given after the /OVERLAYINTERRUPT switch. Try again with a number in the range 4 to 255.

Invalid numeric switch specification
: You made a typographical error entering a value for one of the linker switches (for example, entering a character string for a switch that requires a numeric value). Linker will abort.

Invalid object module
: One of the object modules is invalid. If the error persists, contact Microsoft via the Software Problem Report at the back of this manual.

name is not a valid library
: The file specified as a library is not a valid library file. Linker will abort.

LEDATA record contains more than 1024 bytes of data.
Please note the translator (compiler or assembler) that produced the incorrect object module and the circumstances under which it was produced; report the information to Microsoft via the Software Problem Report at the back of this manual.

Link failed: status (-1)
LINK.EXE could not be found or could not be executed.

Nested left parentheses
You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

No object modules specified
You have failed to supply the linker with any object file names. Try again.

Out of space on list file
Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file
Disk on which ".EXE" is being written is full. Free more space on the disk and try again.

Out of space on scratch file
Disk in default drive is full. Delete some files on that disk, or replace with another disk, and restart the linker.

Overlay manager symbol already defined: *name*
You have defined a symbol name that conflicts with one of the special overlay manager names. Change the offending name and relink.

Please replace original diskette in drive A: and press <ENTER>
This message appears after the ".EXE" has been written if the /P switch is given. Insert the disk with the list file so that it can be reopened.

Relocation table overflow
More than 16,384 long (far) calls, long jumps or other long pointers occur in the user's program. Rewrite program, replacing long references with short references where possible, and recreate object module. NOTE: Pascal and FORTRAN users should first try turning off debugging.

Segment limit set too high
The limit on the number of segments allowed was set too high using the /SEGMENTS switch. Linker will abort.

Segment limit too high
: There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with /SEGMENTS or the default: 128). Either try the link again using /SEGMENTS to select a smaller number of segments (for example, 64, if the default was used previously) or free some memory.

Segment size exceeds 64K
: You have a small model program with more than 64 kilobytes of code, or a medium model program with more than 64 kilobytes of data. Try compiling and linking with the medium or large memory model.

Stack size exceeds 65536 bytes
: The size specified for the stack using the /STACK switch is more than 65,536 bytes. Try again.

Symbol table overflow
: Your program has more than 256 kilobytes of symbolic information (publics, externs, segments, groups, classes, files, etc.). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user
: You entered CONTROL-C, causing the linker to terminate.

Too many external symbols in one module
: Your object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module
: Your program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups
: Your program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module
: Linker encountered more than nine GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries
: You tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many overlays
: Your program defines more than 63 overlays. Reduce the number of overlays.

Too many segments
: Your program has too many segments. Relink using the /SEGMENTS switch with an appropriate number of segments specified.

Too many segments in one module
: Your object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs
: TYPDEFs are records emitted by the compiler to describe communal variables. Create two sources from the old source, dividing the communal variable definitions between them; recompile and relink.

Unexpected end-of-file on library
: The disk containing the library has probably been removed. Try again after inserting the disk containing the library.

Unexpected end-of-file on scratch file
: Disk containing VM.TMP was removed. Restart linker.

Unmatched left parenthesis
: You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

Unmatched right parenthesis
: You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

Unrecognized switch error:
: You entered an unrecognized character after the switch indicator (/), such as LINK /ABCDEF BETA; linker will abort.

VM.TMP is an illegal file name and has been ignored
: You have used VM.TMP as an object file name. Rename file and link again.

Warning: no stack segment
: Your program contains no segment of combine-type stack.

Warning: too many local symbols
: You asked for a sorted listing of local symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the local symbols.

Warning: too many public symbols
: You asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

E.5 Library Manager Error Messages

The following are Microsoft LIB error messages.

symbol is a multiply defined PUBLIC. Proceed?
: Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol. To correct the source, remove the PUBLIC declaration from one of the object modules and recompile or reassemble. If you respond "no", the library will be left in an indeterminate state.

Allocate error on VM.TMP
: Out of disk space. Make space on the disk or in the directory.

Cannot create extract file
: No room in directory for extract file. Make space on the disk or in the directory.

Cannot create list file
: No room in directory for library file. Make space on the disk or in the directory.

Cannot nest response file
: @*filespec* within response file.

Cannot write library file
: Out of disk space. Make space on the disk or in the directory.

Close error on extract file
: Out of disk space. Make space on the disk or in the directory.

Error: An internal error has occurred
: Contact Microsoft Corporation via the Software Problem Report at the back of this manual.

Fatal Error: Cannot open input file
: You mistyped an object filename.

Fatal Error: Module is not in the library
: You tried to delete a module that is not in the library.

Input file read error
: Bad object module or faulty disk.

Invalid object module/library
: Bad object module and/or library.

Library Disk is full
: No more room on disk. Make space on the disk or in the directory.

Listing file write error
: Out of disk space. Make space on the disk or in the directory.

MS-LIB cannot open VM.TMP
: No room for VM.TMP in disk directory. Make space on the disk or in the directory.

No library file specified
: No response to "Library name" prompt.

Read error on VM.TMP
: Disk not ready for read.

Symbol table capacity exceeded
: Too many public symbols (the limit is approximately 30 kilobytes in symbols).

Too many object modules
: More than 500 object modules.

Too many public symbols
: 1,024 public symbols maximum.

Write error on library/extract file
: Out of disk space. Make space on the disk or in the directory.

Write error on VM.TMP
: Out of disk space. Make space on the disk or in the directory.

E.6 EXEPACK Error Messages

The EXEPACK utility generates the following error messages.

exepack: can't change load-high program
When the minimum allocation value and the maximum allocation value are both zero, the file cannot be compressed.

exepack: error reading relocation table
The file cannot be compressed because the relocation table cannot be found or is invalid.

exepack: invalid .EXE file (actual length < reported)
The second and third fields in the file header indicate a file size greater than the actual size.

exepack: invalid .EXE file (bad header)
The given file is not an executable file or has an invalid file header.

exepack: *filename*: No such file or directory
The given file cannot be found.

exepack: *filename*: Permission denied
The given file is a read-only file.

exepack: out of memory
The EXEPACK utility does not have enough memory to operate.

exepack: too many segments in relocation table
The given file is too large to be compressed in the available system memory.

usage: exepack infile outfile
The EXEPACK command line was not specified properly.

You may also encounter MS-DOS error messages if the EXEPACK program cannot read, write to, or create a file.

E.7 EXEMOD Error Messages

The EXEMOD utility generates the following error messages.

exemod: can't change load-high program
: When the minimum allocation value and the maximum allocation value are both zero, the file cannot be modified.

exemod: file not .EXE: *filename*
: EXEMOD automatically appends the ".EXE" extension to any filename without an extension; in this case, no file with the given name and an ".EXE" extension could be found.

exemod: invalid .EXE file (actual length < reported)
: The second and third fields in the file header indicate a file size greater than the actual size.

exemod: invalid .EXE file (bad header)
: The given file is not an executable file or has an invalid file header.

exemod: min > max (correcting max)
: If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. (Note: this is a warning message only; the modification is still performed.)

exemod: min < stack (correcting min)
: If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. (Note: this is a warning message only; the modification is still performed.)

exemod: *filename*: No such file or directory
: The given file cannot be found.

exemod: *filename*: Permission denied
: The given file is a read-only file.

usage: exemod file [-/h] [-/stack n] [-/max n] [-/min n]
: The EXEMOD command line was not specified properly.

The EXEMOD utility also produces error messages when the file header is not in recognizable ".EXE" format, or if errors occur in reading or writing to a file.

Appendix F
Working with Microsoft Products

F.1 Introduction 267

F.2 Microsoft LINK, Microsoft LIB, EXEPACK, and EXEMOD 267

F.3 286 XENIX Operating System 268

F.4 Microsoft FORTRAN and Microsoft Pascal 268

F.5 Microsoft Macro Assembler (MASM) and Symbolic Debug Utility (SYMDEB) 268

F.5.1 SYMDEB Procedures 269

F.5.2 Sample SYMDEB Session 269

F.1 Introduction

This appendix gives an overview of how the Microsoft C Compiler works with other Microsoft languages and tools. In particular, Section F.5 discusses how the compiler works with SYMDEB, the Microsoft Symbolic Debug Utility.

F.2 Microsoft LINK, Microsoft LIB, EXEPACK, and EXEMOD

The programs that accompany the Microsoft C Compiler (the linker, library manager, EXEPACK utility, and EXEMOD utility) work with other Microsoft languages as well as with C. Once you become familiar with the procedures for running these programs, you can use them with other Microsoft languages without having to learn a new program. However, when using these tools, you should always consult the appropriate language documentation to find out what version of the program you have and to learn about any special requirements that may apply when using the tools with a particular language.

The Microsoft Object Code Linker (LINK), which accompanies the C compiler, is the same linker used with all Microsoft compilers, and it offers a wide variety of options to suit the needs of programmers in different languages.

The Microsoft Library Manager (LIB) also accompanies the C compiler. It is used to create and maintain libraries of object files. The library manager accepts MS-DOS object files produced by any Microsoft C compiler. In addition, LIB accepts 286 XENIX archives and Intel-style libraries and allows you to merge these libraries with MS-DOS libraries.

The EXEPACK and EXEMOD utilities (provided with the C compiler) can be used on MS-DOS executable files in any Microsoft language, not just C program files. However, the EXEMOD utility should be used with care, and you should read the discussion of EXEMOD in Section 7.9.2 of Chapter 7, "Advanced Topics," for information about the constraints that apply to executable files in different languages.

F.3 286 XENIX Operating System

The Microsoft C Compiler for MS-DOS shares its design with the C compiler for 286 XENIX systems, and was developed to facilitate portability of C programs between XENIX and MS-DOS systems. If you are interested in writing programs that can be ported to XENIX systems, refer to the *Microsoft C Run-Time Library Reference,* which contains information on portability between the MS-DOS and XENIX C run-time libraries.

F.4 Microsoft FORTRAN and Microsoft Pascal

Versions 3.0 and later of the C compiler allow you to declare and call routines written in Microsoft FORTRAN or Microsoft Pascal. To do this you must have Version 3.3 or later of the Microsoft FORTRAN or Pascal compilers. Section 8.2 of Chapter 8, "Interfaces with Other Languages," describes the syntax for declaring Pascal and FORTRAN routines; however, before attempting to use this feature, you should read the discussion of mixed language programming provided with the Microsoft FORTRAN or Microsoft Pascal Compiler, Version 3.3 or later.

F.5 Microsoft Macro Assembler (MASM) and Symbolic Debug Utility (SYMDEB)

The Microsoft Macro Assembler (MASM) Version 3.0 and later can be used to create assembly language routines to work with C programs. The listing file output by the /Fa option of the C compiler is suitable for input to MASM. See Section 8.1 of Chapter 8, "Interfaces with Other Languages," for a discussion of the assembly language interface to C programs.

The symbolic debug utility (SYMDEB) provided with the Macro Assembler Version 3.0 or later can also be used to debug C programs. SYMDEB can access program locations through addresses, global symbols, or line number references, making it easy to locate and debug specific sections of code. See your Macro Assembler (MASM) documentation for a full description of SYMDEB's capabilities. An introduction to using SYMDEB with C source files is given below.

F.5.1 SYMDEB Procedures

To use SYMDEB with C programs, follow these procedures.

1. When you compile a C source file, use the /Zd and /Od options. The /Zd option produces line numbers in the object file and the /Od option disables optimization so that your code will not be rearranged. (Note: these options are not mandatory for debugging, but they are usually helpful.)
2. When you link, use the /MAP and /LINENUMBERS options to produce a map file that contains line number information.
3. Convert the resulting map file (.MAP) to a symbol file (.SYM) using MAPSYM. MAPSYM is described in the SYMDEB section of the Macro Assembler documentation.
4. Invoke SYMDEB with the symbol file and executable file as arguments (followed by any arguments to the executable file).

When using SYMDEB to debug C programs, you should keep in mind that the C compiler automatically adds a leading underscore to the beginning of every global name. Be sure to include the leading underscore when you specify global symbol names to SYMDEB.

You should also remember that all C programs are linked with a start-up routine (CRT0.OBJ) from the standard C library for the appropriate model. Program execution actually begins with the start-up routine, which then calls the program's "main" function. When you first invoke SYMDEB, the current position will be the beginning of the program, corresponding to the beginning of the start-up routine. Thus, after you invoke SYMDEB, you will probably want to give a command to move to the beginning of the "main" function and start your debugging there.

F.5.2 Sample SYMDEB Session

This sample session with SYMDEB gives examples of some commonly used SYMDEB instructions and shows how to display lines from your source file. For a complete list of SYMDEB options, see your Macro Assembly (MASM) documentation.

A sample program for use in the SYMDEB session is listed below. The sample program contains two functions, *main* and *add,* which are stored in two separate source files named MAIN.C and ADD.C.

```
                    /* CONTENTS OF MAIN.C */

double f = 32.5, g = 16.2;

main()
{
        double a;
        double add(double, double);

        a = add(f, g);
}

                    /* CONTENTS OF ADD.C */
double add(x, y)
double x, y;
{
        return (x + y);
}
```

Give the following commands to compile and link the files.

```
MSC /Zd /Od ADD.C;
MSC /Zd /Od MAIN.C;

LINK /MAP /LINE ADD.OBJ+MAIN.OBJ;
```

You now have a map file (ADD.MAP) and an executable file (ADD.EXE). To create a symbol file, type

```
MAPSYM ADD.MAP
```

This command creates a symbol file named ADD.SYM that the debugger uses to access global symbols. To start the debugger, type

```
SYMDEB ADD.SYM ADD.EXE
```

This command invokes SYMDEB and passes it the names of the symbol file and executable file for the program to be debugged. (In this case, the program ADD.EXE does not take any command line arguments; if it did, they would be given after ADD.EXE on the SYMDEB command line.) The SYMDEB prompt is the hyphen character (-); it appears whenever SYMDEB is waiting for instructions from you.

You can get a quick demonstration of some of SYMDEB's features by following the steps in this list.

Step	Command	Task
1.	`?`	Displays a listing of SYMDEB commands.
2.	`G _main`	Executes all lines up to the beginning of the "main" function. The start-up code precedes the "main" function.
3.	`R`	Displays the current values of all registers.
4.	`u`	Lists program lines. By default, the listing is an assembly listing, so the assembler instructions that correspond to your program code are displayed. When you type "u" without any further arguments, lines are displayed starting at the current location, which corresponds to the beginning of the *main* function.
5.	`S&`	Changes the default listing type to a combined source and assembly listing. After you give this command, type "u _add" to produce the listing. The global symbol "_add" specifies the starting point for the listing. SYMDEB prompts you for the name of the appropriate source file (the one containing the *add* function). Type ADD.C in response to the prompt, and the source file lines corresponding to the *add* function will be displayed.
6.	`S+`	Changes the default listing type to just source file lines. Type "u _add" after giving this command and the source file lines for the *add* function are displayed. Since the source file, ADD.C, was already specified, you do not have to give the filename again.
7.	`BP _add`	Sets a break point at the beginning of the *add* function.
8.	`G`	Executes the program up to the break point at *add*.
9.	`DL _f`	Displays the value of the global variable *f* as a **long float**. The bytes corresponding to the value of *f* are displayed, and the value is also given in exponential notation.
10.	`Q`	Exits the debugger.

Appendix G
Microsoft LINK Technical Summary

G.1 Introduction 275
G.2 Alignment of Segments 275
G.3 Frame Addresses 276
G.4 Order of Segments 276
G.5 Combined Segments 277
G.6 Groups 278
G.7 Fix-ups 279
G.8 Controlling the Loading Order 280

G.1 Introduction

This appendix provides a summary of the operation of the Microsoft LINK linking loader. LINK creates an executable file by concatenating a program's code and data segments according to the instructions supplied in the original source files. These concatenated segments form an "executable image" which is copied directly into memory when you invoke the program for execution. Thus, the order and manner in which LINK copies segments to the executable file defines the order and manner in which the segments will be loaded into memory.

You can tell LINK how to link a program's segments by using command line options with the Microsoft C compiler or by using SEGMENT and GROUP directives in the Microsoft Macro Assembler (MASM). See Section 8.1.1 of Chapter 8, "Interfaces with Other Languages," for a discussion of the segment model for C programs and for a listing of class names, align types, and combine types.

The following sections explain the process LINK uses to concatenate segments and resolve references to items in memory.

G.2 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are BYTE, WORD, PARA, and PAGE. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively.

When LINK encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is WORD, PARA, or PAGE, LINK checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, LINK pads the image with extra zero bytes.

G.3 Frame Addresses

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the size of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number."

The canonical frame number for a particular segment can be determined from the list file produced by LINK as follows. The list file gives the starting address for each segment as a five-digit hexadecimal number. The four most significant digits taken as a four-digit hexadecimal number give the canonical frame number for the segment. The offset is given by the least significant digit of the starting address. The frame number selects a particular paragraph. A paragraph is a set of 16 contiguous bytes such that the address of the first byte in the set is a multiple of 16. The offset is the number of bytes from the first byte in the paragraph to the first byte in the segment.

For BYTE and WORD alignments, the offset may be nonzero. The offset is always zero for PARA and PAGE alignments.

G.4 Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless LINK encounters two or more segments having the same class name. Segments having identical class names belong to the same class and are copied as a contiguous block to the executable file.

For example, in the following program fragment the segments "DATAX" and "DATAZ" form a class. Both segments are copied to the executable file before the "TEXT" segment.

```
DATAX   segment 'DATA'
DATAX   ends

TEXT    segment 'CODE'
TEXT    ends

DATAZ   segment 'DATA'
DATAZ   ends
```

All segments belong to a class. Segments for which no class name is explicitly defined have the "null" class name, and will be loaded as a contiguous block with other segments having the null class name.

LINK imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64K (kilobytes).

Note

The Microsoft C Compiler, Versions 3.0 and later, and the Microsoft FORTRAN and Pascal Compilers, Versions 3.3 and later, use the segment ordering specified by the /DOSSEG linker option. This imposes additional constraints on the segment loading order. See the discussion of the /DOSSEG option in Section 4.5.10, "Ordering Segments," of Chapter 4, "Linking," for details.

G.5 Combined Segments

LINK uses combine types to determine whether or not two or more segments sharing the same segment name should be combined into a single large segment. The combine types are "public," "stack," "memory," "common," and "private."

If a segment has public type, LINK will automatically combine it with any other segments having the same name and belonging to the same class. When LINK combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K (kilobytes), LINK displays an error message.

If a segment has stack or memory type, LINK carries out the same combine operation as public segments. The only exception is that stack segments cause LINK to copy an initial stack pointer value to the executable file. This stack pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has common type, LINK will automatically combine it with any other segments having the same name and belonging to the same class. When LINK combines common segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has private type only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

G.6 Groups

Groups let segments that are not contiguous and do not belong to the same class be addressable relative to the same frame address. When LINK encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address. LINK does not check to see if all elements of a group fit within the same 64K of memory.

Segments in a group do not have to be contiguous, do not have to belong to the same class, and do not have to have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, LINK may place segments that do not belong to the group in the same 64K of memory. Although LINK does not explicitly check that all segments in a group fit within 64K of memory, LINK is likely to encounter a fix-up overflow error if this requirement is not met.

G.7 Fix-ups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, LINK can "fix up" any unresolved references to labels and variables. To fix up unresolved references, LINK computes an appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fix-ups for four different references:

- Short
- Near self-relative
- Near segment-relative
- Long

The size of the value to be computed depends on the type of reference. If LINK discovers an error in the anticipated size of a reference, it displays a fix-up overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction in a segment having a different frame address. It can also occur if all segments in a group do not fit within a single 64K block of memory.

A short reference occurs in JMP instructions that attempt to pass control to labeled instructions that are in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference. LINK computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (has a different frame address) or the target is more than 128 bytes distant (either direction).

A near self-relative reference occurs in instructions which access data relative to the same segment or group. LINK computes a 16-bit offset for this reference. It displays an error if the data is not in the same segment or group.

A near segment-relative reference occurs in instructions that attempt to access data in a specified segment or group or relative to a specified segment register. LINK computes a 16-bit offset for this reference. It displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0 or if the beginning of the canonical frame of the target is not addressable.

A long reference occurs in CALL instructions that attempt to access an instruction in another segment or group. LINK computes a 16-bit frame address and 16-bit offset for this reference. LINK displays an error message if the computed offset is greater than 64K or less than zero or if the beginning of the canonical frame of the target is not addressable.

G.8 Controlling the Loading Order

You can control the loading order of the segments in a program by creating and assembling a dummy program file that contains empty segment definitions given in the order you wish to load your real segments. Once this file is assembled, you simply give it as the first object file in any invocation of LINK. LINK will automatically load the segments in the order given.

For example, the following dummy program file defines the loading order of segments in a program having segments named CODE, DATA, STACK, CONST, and MEMORY.

```
CODE    segment 'CODE'
CODE    ends
CONST   segment 'CONST'
CONST   ends
DATA    segment 'DATA'
DATA    ends
STACK   segment stack 'STACK'
STACK   ends
MEMORY  segment 'MEMORY'
MEMORY  ends
```

The dummy program file must contain definitions for all classes to be used in your program. If it does not, LINK will choose a default loading order, which may or may not correspond to the order you desire. When linking your program, the dummy program must be the first object file specified in the LINK command line.

Important

Do not use a dummy program file with C, Pascal, or other high-level language programs. These languages define their own loading order. This order must not be modified.

You can force LINK to load MEMORY segments as the last segments in a program by placing an empty MEMORY segment at the end of your dummy program file. The empty segment should have the form

segment-name SEGMENT MEMORY '*class-name*'
segment-name ENDS

where *segment-name* is the name you intend to use for MEMORY segments and *class-name* is the name you intend to use for the memory class.

Example

```
MEMORY segment memory 'MEMORY'
MEMORY ends
```

Index

User's Guide Index

80186/80188 processor, 24, 68
80286 processor, 24, 68
8087/80287 coprocessor, 63, 64
  controlling use, 65, 143
  in-line instructions, 64, 65
87.LIB, 18, 64, 141

$ (dollar sign), 202
& (ampersand), 125, 129
* (asterisk)
  as LIB command symbol, 126, 132
  as wild card character, 19, 114
  in CL command, 191
+ (plus sign)
  as LIB command symbol, 125, 131
  in LINK command, 91
, (comma), 46
- (hyphen) option character, 48, 192
– (minus sign), 126, 132
–* (minus-asterisk), 126, 132
–+ (minus-plus), 126, 132
/ (forward slash) option character
  LINK, 94
  MSC, 48
; (semicolon)
  in LIB command, 125, 128, 130
  in LINK command, 91
  in MSC command, 45
? (question mark)
  as wild card character, 19, 114
  in CL command, 191
_ (underscore)
  in global names, 219
  in identifiers, 168

/A options, 76, 148, 149, 151, 152
abs, 208
Adding an object module to a library, 125, 131
/AL option, 79
Alias checking, 73
  in previous versions of the compiler, 205
Align types, 163, 275
allmem, 207
Allocating paragraph space, 102
Alternate math library, 18, 66, 141
/AM option, 79
Ampersand (&), 125, 129
argc variable, 112
Argument type checking, 61
Arguments
  command line, 115
  conversion, 165
  pushing, 165
  to /F options, 49, 193
  to /Gt option, 49
  to LINK options, 95
  to macros, 254
  to main function. *See* main function.
  to MSC options, 49
  wild card, 114
argv variable, 112
Array
  identifiers, 203
  limits, 204
/AS option, 79
ASCII character codes, 175
Assembly language interface, 159
  in previous versions of the compiler, 212
Assembly listing file, 52
Assertion count exceeds 5; stopping compilation, 231
Assertion error messages, 231
Asterisk (*)
  as LIB command symbol, 126, 132
  as wild card character, 19, 114
  in CL command, 191
AUTOEXEC.BAT file, 22
AUX, 43

Backing up disks, 13
Batch files, 22, 36
BEGDATA class name, 105
Bibliography, 9
Binary mode, 19, 147
BINMODE.OBJ, 19, 147
Bitfields, 203
BP register, 165, 167, 214
Brackets, 6
BSS class name, 105
_BSS segment, 161
Buffers, 23
BYTE align type, 275

/C option, 59
/c option, 195, 196
Calling conventions, 165
  in previous versions of the compiler, 214
Calling FORTRAN and Pascal routines, 170
Canonical frame number, 276
Capital letters, 7, 8
Carriage return-linefeed (CR-LF) translation, 147
Case significance
  assembly language interface, 168
  filenames, 42
  in previous versions of the compiler, 219
  LINK, 88, 99
  MSC options, 49
Changing libraries at link time, 141
char constants, 202
char type, 202
Checking syntax, 60
_chkstk, 217
chksum, 116
CL command, 41, 87, 191
  /c option, 195, 196
  /F options, 192
  /Fe option, 192, 196
  /Fm option, 192, 196
  /link option, 195, 196
  linking, 195
  syntax, 191
  XENIX-compatible options, 196
CL.EXE, 16, 17
Class names, 163
  BEGDATA, 105
  BSS, 105
  CODE, 105
  FAR_BSS, 161
  FAR_DATA, 161
  STACK, 105
Classes, 163, 277
  in previous versions of the compiler, 222
CODE class name, 105
Code pointers, 151
Code segments, 162
  naming, 155
  restrictions, 75
Code size, 73
Combine types, 163, 277
Combined listing file, 52
Combining libraries, 125, 131
Comma (,), 46
Command line
  arguments, 111
  error messages, 69, 230, 251
  executable file, 111
  maximum length, 112, 229
  method
    LIB, 127
    LINK, 92
    MSC, 46
  suppressing processing of, 115
  wild cards, 114
Command symbols
  asterisk (*), 126, 132
  minus (-), 126, 132
  minus-asterisk -*), 126, 132
  minus-plus (-+), 126, 132
  plus (+), 125, 131
Commands
  CL. *See* CL command.
  IF ERRORLEVEL, 37
  notational conventions, 7
  PATH, 13, 20, 22
  SET, 13, 20, 21, 22
  summary, 179
Comments
  in previous versions of the compiler, 201

Comments *(continued)*
preserving, 59
Common combine type, 277
Communal variables, 203, 204
Compatibility
between floating-point options, 66
with MASM, 268
with Microsoft FORTRAN, 268
with Microsoft Pascal, 268
with SYMDEB, 268
with the 286 XENIX operating system, 268
with XENIX options, 196
Compilation error messages, 69, 230
Compile only option, 195, 196
Compiler differences, 201
alias checking, 205
array identifiers, 203
array limits, 204
assembly language interface, 212
bitfields, 203
char constants, 202
char type, 202
comments, 201
enum type, 202
equality operators, 203
function identifiers, 203
identifiers, 202
language definition, 201
logical AND and OR operators, 205
lvalue expressions, 203
macro definitions, 204
preprocessor, 204
relational operators, 203
run-time library, 206
strings, 202
structure identifiers, 203
structures and unions, 204
type casts, 203
uninitialized variables at external level, 203
union identifiers, 203
unsigned long type, 202
Compiler error messages, 230
assertion, 231
code generation, 231
command line, 230, 251
compilation, 230
Compiler error messages *(continued)*
fatal, 230, 239
internal, 230
warning, 230, 232
Compiler exit codes, 72
Compiler limits, 253
Compiler naming conventions, 53, 98
Compiler options. *See* MSC options.
Compiler passes, 16
Compiler summary, 179
Compiling large programs, 75
Compressing executable files, 145
CON, 43
Conditional compilation, 54
CONFIG.SYS file, 23
Consistency check, 125, 134
CONST segment, 161
Constants
defining, 54
maximum size, 254
Control program, 16
CONTROL-C, 42, 92, 129
Conversions, 165
Converting from previous versions of the compiler
assembly language interface, 212
compiler differences, 201
run-time library differences, 206
Coprocessor, 24, 63
suppressing use of, 143
cparMaxALLOC field, 102
/CPARMAXALLOC option, 102
CR-LF (carriage return-linefeed) translation, 147
creat, 208
Cross-reference listing (LIB), 126, 133
CRT0.OBJ, 18, 101
CS. *See* Code segments.
CS register, 165, 169
Customized memory models, 149
c_common segment, 161

/D option, 54
Data files, 19
Data loading, 106
Data passed at execution, 111
Data pointers, 152

Data segment, 152
  default, 154, 161
  default name, 155
  naming, 155
  restrictions, 75
_DATA segment, 155, 161
Data threshold, 154
Debugging
  preparation for, 72
  procedures (SYMDEB), 264
Declarations, 254
Default data segment, 154, 161
Default file extensions
  LINK, 88
  MSC, 42
Default libraries, 18, 85, 90
  changing, 86
  ignoring, 100
  order of linking, 141
  overriding at link time, 141
  search path, 85, 90
  suppressing selection, 139
Default responses
  LIB, 130
  LINK, 91
  MSC, 45
defined(identifier) constant-expression, 204
Defining constants and macros, 54
Deleting an object module from a library, 126, 132
Denormal numbers, 66, 227
Device names, 43
DGROUP, 105, 162
DI register, 165, 167, 169, 213
Differences. *See* Converting from previous versions of the compiler.
Direction flag, 169, 213
Directory names, 7
Disabling optimization, 72, 73
Disks
  backing up, 13
  contents, 13
  organizing, 25
  swapping, 45
Displaying line numbers, 99
Dollar sign ($), 202
/DOSSEG option, 105
DS register, 162, 165, 169, 213, 221
DS. *See* Data segment.
/DSALLOCATE, 106

/E option, 57
/EP option, 57
#elif directive, 204
Ellipses, 7
EM.LIB, 18, 63, 64, 65, 141
Emulator
  in-line instructions, 64, 65
  library. *See* EM.LIB.
Enabling special keywords, 137, 150
Entry sequence, 165
  in previous versions of the compiler, 214
enum type, 202
environ variable, 113
Environment table, 113
  maximum size, 229
  suppressing processing of, 115
Environment variables, 13, 20, 179
  defining, 21
  notational conventions, 7
  overriding, 23
Environment
  changing, 23
  setting up, 20
    with batch files, 36
envp variable, 112
/EP option, 57
Equality operators, 203
Error messages, 225
  compiler, 68, 230
    assertion, 231
    command line, 69, 251
    compilation, 69, 230
    fatal, 69, 239
    internal, 230
    warning, 69, 230, 232
  EXEMOD, 264
  EXEPACK, 263
  floating-point exceptions, 227
  library manager, 261
  linker, 255
  run-time, 225
ES register, 162, 213, 221

Exceptions, 227
Exclude option, 60
Executable files, 13, 16, 111
  command line arguments, 111
  compressing, 145
  modifying, 145
  naming, 89, 192
  passing data to, 111
  search path, 20, 21
Executable image, 83
Executing programs, 111
Execution time, 73
EXEMOD, 17, 145, 146
  compatibility with other Microsoft products, 267
  error messages, 264
  /max option, 146
  /min option, 146
  /stack option, 146
  summary, 188
EXEPACK, 17, 145, 146
  compatibility with other Microsoft products, 267
  error messages, 263
  summary, 187
Exit codes, 72
Exit sequence, 167
  in previous versions of the compiler, 214
Extending lines, 91, 125, 129
Extensions, 42, 88
External names, 138
Extracting an object module from a library, 126, 132

/F options, 50, 52, 192
  arguments, 49, 193
  in CL command, 192
/Fa option, 52
far keyword, 137, 148, 149
Far pointers, 78, 148
FAR_BSS, 161
FAR_DATA, 161
Fatal error messages, 69, 230, 239
/Fc option, 52
/Fe option, 192, 196
Filename extensions
Filename extensions *(continued)*
  conventions
    LINK, 88
    MSC, 42
  extensions, 42, 88
  notational conventions, 7
  special, 43
Files
  batch. *See* Batch files.
  data. *See* Data files.
  executable. *See* Executable files.
  in CONFIG.SYS file, 23
  include. *See* Include files.
  library. *See* Library files.
  locating, 20
  maximum number open, 229
  maximum size, 229
  temporary. *See* Temporary files.
Fix-ups, 279
/Fl option, 52
Floating-point error, 227
Floating point not loaded, 64, 226
Floating-point libraries, 18, 63
  alternate math, 66
Floating-point operations, 63, 140, 141
  compatibility, 66
  default, 65
  error messages, 227
  exceptions, 227
  function calls, 64, 65
  in-line instructions, 64, 65
  maximum efficiency, 64, 66
  maximum flexibility, 66
  maximum precision, 64, 66
Floating-point options, 63
/Fm option, 192, 196
/Fo option, 50
fopen, 209
fortran keyword, 137, 170
FORTRAN routines, 268
  declaring, 170
Forward slash (/)
  as LINK option character, 94
  as MSC option character, 48
/FPa option, 63, 66, 141
/FPc option, 63, 65, 141
/FPc87 option, 63, 64, 141
/FPi option, 63, 64, 65, 141

/FPi87 option, 63, 64, 141
Frame addresses, 276
Frame number, 276
Frame pointer, 214
freopen, 209
Function declarations, 61
Function identifiers, 203

/G0 option, 68
/G1 option, 68
/G2 option, 68
Generating function declarations, 61
getenv, 113
getmem, 207
Global symbols. *See* Public symbols.
Global variables, 219
Groups, 162, 278
  DGROUP, 105, 162
  in previous versions of the compiler, 222
/Gs option, 144
/Gt option, 154
  arguments, 49

/H option, 138
Heap, 160
Helvetica font, 8
/HIGH option, 106
huge keyword, 137
Hyphen, 48

/I option, 59
Identifiers
  array, 203
  function, 203
  in previous versions of the compiler, 202
  notational conventions, 6
  maximum length, 254
  predefined, 56
    removing definitions, 57
  structure, 203
  union, 203
Identifying syntax errors, 61
IF ERRORLEVEL command, 37, 72
Ignoring case, 99
Ignoring default libraries, 100
Illegal allocation of segment-type segment>64K, 75
In-line instructions, 64, 65
#include directive, 17
Include files, 14, 17
  changing search path for, 60
Include files
  maximum level of nesting, 254
  search path, 20, 21, 59
  standard places, 59
  V2toV3.H, 206
INCLUDE variable, 21, 59
  overriding effect of, 60
Infinities, 66
Installation, 13
Instruction set
  80186/80188 processor, 68
  80286 processor, 68
  8086/8088 processor, 68
Integer size summary, 183
Interfaces with other languages, 159
Internal error messages, 230
iscsym, 210
iscsymf, 210
Italics, 6

Key sequences, 8
Keywords
  notational conventions, 8
  special, 137
Kilobyte, 75

Labeling the object file, 139
Language definition differences, 201
Large model, 77, 79
Large model library files, 17
Large programs, 75
Learning more about C, 9
LIB command, 17
  adding a library module, 125, 131
  backup library file, 123
  command line method, 127
  command symbols, 125
  default responses, 130

LIB command *(continued)*
deleting a library module, 126, 132
error messages, 261
extending lines, 125
extracting a library module, 126, 132
modification methods, 123
order of operations, 122
/pagesize option, 134
prompts, 123
replacing a library module, 126, 132
response file method, 128
setting page size, 134
summary, 187
terminating, 129
LIB variable, 21, 85, 90
Libraries
8087/80287, 141
alternate math, 66, 141
combining, 133
creating, 121, 130
/Zl option, 140
default. *See* Default libraries.
emulator, 141
floating-point, 63
for mixed model programs, 153
modifying, 121, 131
search path, 85, 90
Libraries prompt, 89
Library files
default, 18
floating-point, 18
large model, 17
medium model, 17
search path, 21
small model, 17
standard C, 18
Library listing, 126, 133
Library manager, 17
See also LIB command.
compatibility with other Microsoft products, 267
error messages, 261
Library name prompt, 124
Library page size, 134
Library search path, 85, 90
Limits
compiler, 253
run-time, 229
Line number option, 72
Line numbers
displaying in linker listing file, 99
/LINENUMBERS option, 99
LINK command, 17, 20
default responses, 91
extending lines, 91
prompts, 87
separating entries, 91
terminating, 92
/link option, 195, 196
LINK options, 94
/CPARMAXALLOC, 102
/DOSSEG, 105
/DSALLOCATE, 106
/HIGH, 106
/LINENUMBERS, 99
/MAP, 97
/NOD, 67, 141
/NODEFAULTLIBRARYSEARCH (/NOD), 100
/NOGROUPASSOCIATION, 107
/NOIGNORECASE (/NOI), 99
/OVERLAYINTERRUPT, 105
/PAUSE, 95
/SEGMENTS, 102
/STACK, 101
abbreviations, 94
allocating paragraph space, 102
controlling data loading, 106
controlling run file loading, 106
controlling segments, 102
controlling stack size, 101
displaying line numbers, 99
ignoring default libraries, 100
map file, 97
no default library search, 67, 141
no ignore case, 99
numerical arguments, 95
ordering segments, 105
pausing, 95
preserving compatibility, 107
setting the overlay interrupt, 105
summary, 185
with C programs, 86
Linker, 17
See also LINK command
compatibility with other Microsoft products, 267

Linker *(continued)*
error messages, 255
operation, 83
summary, 184
technical summary, 275
Linking
C program files, 84
overriding default libraries, 141
with CL command, 195
LINT_ARGS, 62
List File prompt, 89, 126, 133
Listing files
assembly listing, 52
combined listing, 52
LIB, 122, 126, 133
LINK, 89, 96
object listing, 52
preprocessed listing, 57
with public symbols, 97
LLIBC.LIB, 19
LLIBFA.LIB, 19, 66, 141
LLIBFP.LIB, 19, 64, 141
Loading order, 280
Logical AND and OR operators, 205
Long pointers. *See* Far pointers.
Long references, 279
LSETARGV.OBJ, 19, 114
Lvalue expressions, 203

Macro Assembler. *See* MASM
Macros
defining, 54
maximum number of arguments, 254
maximum size, 254
notational conventions, 7
main function, 85
arguments to, 111
Manifest constants, 7
defining, 54
maximum size, 254
notational conventions, 7
Map file, 97
naming, 192
/MAP option, 97
MASM (Macro Assembler), 268
max, 210
/max option, 146
Maximum allocation value, 146
Maximum file size, 229
Maximum length of a string, 254
Maximum length of an identifier, 254
Maximum length of command line, 229
Maximum length of preprocessor argument, 254
Maximum level of nesting
declarations, 254
preprocessor directives, 254
Maximum number of macro arguments, 254
Maximum number of open files, 229
Maximum optimization, 145
Maximum size of constant, 254
Maximum size of environment table, 229
Maximum size of macro definition, 254
Medium model, 77, 79
Medium model library files, 17
Memory model options, 76, 149
code pointer size, 151
data pointer size, 152
segment setup, 152
standard memory models, 79
Memory models
customized, 149
default, 79
large, 17, 79
medium, 17, 79
mixed, 148, 149, 151, 152
library support, 153
small, 17, 79
standard, 76
summary, 183
Microsoft LIB. *See* LIB command.
Microsoft LINK. *See* LINK command.
Microsoft products, 267
min, 210
/min option, 146
Minimum allocation value, 146
Minus sign (-), 126, 132
Minus-asterisk (-*), 126, 132
Minus-plus (-+), 126, 132
Mixed memory models, 148, 149, 151, 152
library support, 153
MLIBC.LIB, 19

MLIBFA.LIB, 18, 66, 141
MLIBFP.LIB, 19, 64, 141
Modifying the executable file, 145
Modules, 155
  See also Object modules.
movmem, 210
MSC command
  default responses, 45
  partial command line, 47
  prompts, 42
  responding to prompts, 42
  using the command line, 46
MSC options, 48
  /A, 76, 148, 151, 152
  /AL, 79
  /AM, 79
  /AS, 79
  /C, 59
  /D option, 54
  /E, 57
  /EP, 57
  /Fa option, 52
  /Fc option, 52
  /Fl option, 52
  /Fo, 50
  /FPa, 63, 66, 141
  /FPc, 63, 65, 141
  /FPc87, 63, 64, 141
  /FPi, 63, 64, 65, 141
  /FPi87, 63, 64, 141
  /G0, 68
  /G1, 68
  /G2, 68
  /Gs, 144
  /Gt, 154
  /H, 138
  /I, 59
  /ND, 155
  /NM, 155
  /NT, 155
  /O, 73
  /Od, 72
  /Ox, 145
  /P, 57
  /U, 57
  /u, 57
  /V, 139
  /W, 70
MSC options *(continued)*
  /w, 70
  /X, 59
  /Zd, 72
  /Ze, 137, 150
  /Zg, 61
  /Zl, 139
  /Zp, 137
  /Zs, 61
  arguments to, 49
  assembly listing, 52
  case of, 49
  combined listing, 52
  defining constants and macros, 54
  disabling optimization, 72
  enabling special keywords, 137, 150
  floating-point, 63, 64, 65, 141
  generating function declarations, 61
  identifying syntax errors, 61
  labeling the object file, 139
  line numbers, 72
  memory model
    code pointer size, 151
    customized, 149
    data pointer size, 152
    large, 79
    medium, 79
    mixed, 148, 151, 152
    setting up segments, 152
    small, 79
    standard, 76
  naming data segments, 155
  naming modules, 155
  naming text segments, 155
  object listing, 52
  optimizing, 73, 144, 145
  packing structure members, 137
  preprocessed listing, 57
  preprocessor, 54, 57, 59
  preserving comments, 59
  removing definitions of predefined identifiers, 57
  removing stack probes, 144
  restricting length of external names, 138
  searching for include files, 59
  setting the data threshold, 154
  setting warning level, 70

MSC options *(continued)*
spaces in, 49
summary, 180
suppressing default library selection, 139
using 80186/80188 and 80286 processors, 68
XENIX-compatible, 196
MSC.EXE, 16, 20
MSDOS, 56
MSETARGV.OBJ, 19, 114
M_I86, 56
M_I86xM, 56

Naming conventions
compiler, 53, 98, 168
in previous versions of the compiler, 219
segments, 155
Naming modules, 155
Naming segments, 155
Naming the executable file, 89, 192
Naming the map file, 192
Naming the object file, 50
NaN's, 66
near keyword, 137, 148, 149
/ND option, 155
Near pointers, 78, 148
Near segment-relative references, 279
Near self-relative references, 279
Nesting
declarations, 254
include files, 254
preprocessor directives, 254
/NM option, 155
No default library search option, 67
NO87 variable, 143
/NOD option, 67, 100, 141
/NODEFAULTLIBRARYSEARCH option, 67, 100, 141
/NOGROUPASSOCIATION option, 107
/NOI option, 99
/NOIGNORECASE option, 99
Notational conventions, 6
/NT option, 155
NUL, 43, 127, 133
NUL.MAP, 89
Null pointer assignment, 116, 226
NULL segment, 116, 161, 226
_nullcheck, 116

/O option, 73
Object file, 121
labeling, 139
naming, 50
Object filename prompt, 44
Object listing file, 52
Object listing prompt, 44
Object modules, 121
Object Modules prompt, 88
/Od option, 72
open, 211
Operations prompt, 125
Operators
equality, 203
logical AND and OR, 205
relational, 203
Optimization, 73, 145
advanced, 143
default, 74
disabling, 72, 73
favoring code size, 73
favoring execution time, 73
maximum, 145
options, 73
relaxing alias checking, 73
removing stack probes, 144
Optional fields, 6
Options
LINK. *See* LINK options.
MSC. *See* MSC options.
summary, 179
Ordering segments, 105
Organizing files
floppy disk system, 27
hard disk system, 26
Output library prompt, 127
Overflow, 228
/OVERLAYINTERRUPT option, 105
Overlays, 103
overlay manager prompts, 104
setting the interrupt number, 105
Overview, 3

/Ox option, 145
O_BINARY, 209
O_RAW, 209

/P option, 57
P0.EXE, 16, 20
P1.EXE, 16, 20
P2.EXE, 16, 20
P3.EXE, 16, 20
Packing structure members, 137
PAGE align type, 275
/pagesize option, 134
Page size, 134
PARA align type, 275
Paragraph, 276
Paragraph space, 102
pascal keyword, 137, 170
Pascal routines, 170
Passing data at execution, 111
PATH command, 13, 20, 22
  in AUTOEXEC.BAT file, 22
  in batch files, 22
PATH variable, 21, 22, 111
Pathnames, 7
/PAUSE option, 95
Pausing during linking, 95
peek, 207
Plus sign (+)
  in LIB command, 125, 131
  in LINK command, 91
Pointers
  code. *See* Code pointers.
  data. *See* Data pointers.
  far. *See* Far pointers.
  near. *See* Near pointers.
  summary of sizes, 183
poke, 207
Practice session, 31
Precision, 227
Predefined identifiers, 56
  removing definitions, 57
Preparing for debugging, 72
Preprocessed listing file, 57
Preprocessor options, 54
  defining constants and macros, 54
  preserving comments, 59
  producing listings, 57
Preprocessor options *(continued)*
  removing definitions of predefined identifiers, 57
  searching for include files, 59
Preprocessor
  #elif directive, 204
  defined(identifier) constant-expression, 204
  in previous versions of the compiler, 204
  maximum level of nesting, 254
  maximum number of macro arguments, 254
  maximum size of macro definition, 254
Preserving comments, 59
Private combine type, 277
PRN, 43
Processors
  80186/80188, 24, 68
  80286, 24, 68
  8086/8088, 24, 68
Producing listing files
  assembly listing, 52
  combined listing, 52
  object listing, 52
  preprocessed listings, 57
Public combine type, 277
Public names. *See* External names.
Public symbols, 97
Pushing arguments, 165
putenv, 113

Question mark (?)
  as wild card character, 19, 114
  in CL command, 191
Quotation marks, 8

rbrk, 207
README.DOC, 19
References
  long, 279
  near segment-relative, 279
  near self-relative, 279
  short, 279
Register usage conventions, 169

Register usage conventions *(continued)*
in previous versions of the compiler, 213
Register variables, 213
Registers, 169
BP, 165, 167, 214
CS, 165, 169
DI, 165, 167, 169, 213
DS, 162, 165, 169, 213, 221
ES, 162, 213, 221
SI, 165, 167, 169, 213
SP, 165
SS, 162, 165, 169, 213
Relational operators, 203
Relocation information, 83
Removing definitions of predefined identifiers, 57
Removing stack probes, 144
Replacing an object module in a library, 126, 132
repmem, 207
Response file
LIB, 128
LINK, 93
Restricting length of external names, 138
Return value conventions, 166
rlsmem, 207
rstmem, 207
Run file, 83
See also Executable files.
Run file loading, 106
Run File prompt, 89
Run-time error messages, 225
Run-time library differences, 206
abs, 208
allmem, 207
creat, 208
fopen, 209
freopen, 209
getmem, 207
iscsym, 210
iscsymf, 210
max, 210
min, 210
movmem, 210
open, 211
peek, 207
poke, 207
rbrk, 207
repmem, 207
rlsmem, 207
rstmem, 207
setmem, 211
setnbuf, 211
sizmem, 207
stcarg, 207
stccpy, 207
stcd_i, 207
stch_i, 207
stci_d, 207
stcis, 212
stcisn, 212
stclen, 212
stcpam, 207
stcpm, 207
stcu_d, 207
stpblk, 207
stpbrk, 212
stpchr, 212
stpsym, 207
stptok, 207
stscmp, 212
stspfp, 207
V2toV3.H, 206
Run-time limits, 229
Running programs, 111
Running the compiler
command line method, 46
partial command line, 47
responding to prompts, 42
Running the library manager
command line method, 127
responding to prompts, 123
response file method, 128
Running the linker
command line method, 92
responding to prompts, 87
response file method, 93

Sample setup, 25
floppy disk, 27
hard disk, 26
Search paths, 20

Search paths *(continued)*
changing, 23
executable files, 20, 21
include files, 20, 21, 59
libraries, 20, 21, 85, 90
Segment model, 152, 159
in previous versions of the compiler, 219
Segment naming conventions, 155
in previous versions of the compiler, 219, 222
summary, 184
Segment order, 105, 159, 276
Segments, 159
align type, 163, 275
_BSS, 161
class name, 163
code, 155, 162
combine class, 163, 277
CONST, 161
c_common, 161
_DATA, 161
data, 152, 154, 155, 161
loading order, 280
naming, 155
NULL, 116, 161, 226
number allowed, 102
setting up, 152
stack, 152
STACK, 160
_TEXT, 162
text, 155, 162
/SEGMENTS option, 102
Semicolon (;)
LIB command, 125, 128, 130
LINK command, 91
MSC command, 45
SET command, 13, 20, 21, 22
_setargv, 115
setenvp, 115
setmem, 211
setnbuf, 211
Setting up the environment, 20
Setting up segments, 152
Short pointers. *See* Near pointers.
Short references, 279
SI register, 165, 167, 169, 213
sizmem, 207
SLIBC.LIB, 18
SLIBFA.LIB, 18, 66, 141
SLIBFP.LIB, 18, 64, 141
Small capitals, 8
Small model, 77, 79
Small model library files, 17
Source filename prompt, 44
SP register, 146, 165, 217
setting initial value, 146
Special filenames, 43
Special keywords, 150
enabling, 137
SS register, 162, 165, 169, 213
SSETARGV.OBJ, 19, 114
SS_NE_DS, 56
Stack checking, 144
in previous versions of the compiler, 217
STACK class name, 105
Stack combine type, 277
/STACK option, 101
/stack option, 146
Stack order, 165
Stack overflow, 227
Stack pointer. *See* SP register.
Stack probes, 144
Stack segment, 152
STACK segment, 160
Stack setup, 152, 159
in previous versions of the compiler, 214
Stack size, 101
Standard C library, 18
Standard memory models, 76, 183
Standard places, 20
changing, 60
ignoring, 60
include files, 59
libraries, 85, 90
Standard search paths., 20
See also Search paths.
Start-up routine, 18, 85, 101
stcarg, 207
stccpy, 207
stcd_i, 207
stch_i, 207
stcis, 212
stcisn, 212

stci_d, 207
stclen, 212
stcpam, 207
stcpm, 207
stcu_d, 207
stdargv, 114
Stopping LIB, 129
Stopping LINK, 92
Stopping the compiler, 42
stpblk, 207
stpbrk, 212
stpchr, 212
stpsym, 207
stptok, 207
Strings
  in previous versions of the compiler, 202
  maximum length, 254
  notational conventions, 8
Structures
  in previous versions of the compiler, 203, 204
  packing, 137
stscmp, 212
stspfp, 207
Suppressing command line processing, 115
Suppressing default library selection, 139
Suppressing null pointer checks, 116
Suppressing processing of environment table, 115
Suppressing use of coprocessor, 143
Swapping disks, 45, 95
Switches. *See* Options.
Symbolic constants. *See* Manifest constants.
SYMDEB (Symbolic Debug Utility), 268
Syntax checking, 60
Syntax conventions.
  See Notational conventions.
Syntax errors, 61, 69
SYS subdirectory, 17

Temporary files, 21, 253
Text mode, 19, 147
_TEXT segment, 155, 162
Text segments, 162
  naming, 155
  restrictions, 75
TMP variable, 21
Translation mode, 19, 147
Type casts, 203
Type-checking, 62

/U option, 57
/u option, 57
Underflow, 227
Underscore (_), 53, 98, 168, 219
Uninitialized variables, 161
  in previous versions of the compiler, 203
Unions, 203, 204
unsigned long type, 202

/V option, 139
V2TOV3.H, 206
Variables
  communal, 203, 204
  environment. *See* Environment variables.
  global, 168, 219
  register, 213
  uninitialized, 161, 203
VM.TMP, 84

/W option, 70
/w option, 70, 71
Warning messages, 69, 230, 232
  setting level of, 70
WARNING: NO STACK SEGMENT, 101
Wild card arguments, 19, 114
  in CL command, 191
WORD align type, 275

/X option, 59
XENIX compatibility, 268
XENIX-compatible options, 196

/Zd option, 72
/Ze option, 137, 150
/Zg option, 61
/Zl option, 139
/Zp option, 137
/Zs option, 61

Language Reference

Language Reference

Microsoft® C

Language Reference

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985

If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Microsoft, and the Microsoft logo, are registered trademarks, and MS is a trademark of Microsoft Corporation.

Document Number 8416L-300-00

Contents

1 Introduction 1

1.1 Overview 3
1.2 About This Manual 4

2 Elements of C 9

2.1 Introduction 11
2.2 Character Sets 11
2.3 Constants 17
2.4 Identifiers 23
2.5 Keywords 24
2.6 Comments 25
2.7 Tokens 26

3 Program Structure 27

3.1 Introduction 29
3.2 Source Program 29
3.3 Source Files 30
3.4 Program Execution 32
3.5 Lifetime and Visibility 33
3.6 Naming Classes 37

4 Declarations 41

4.1 Introduction 43
4.2 Type Specifiers 44
4.3 Declarators 48
4.4 Variable Declarations 53
4.5 Function Declarations 66
4.6 Storage Classes 69
4.7 Initialization 76
4.8 Type Declarations 80
4.9 Type Names 82

5 Expressions and Assignments 85

5.1 Introduction 87
5.2 Operands 87
5.3 Operators 97
5.4 Assignment Operators 116
5.5 Precedence and Order of Evaluation 119
5.6 Side Effects 123
5.7 Type Conversions 124

6 Statements 131

6.1 Introduction 133
6.2 Break Statement 135
6.3 Compound Statement 136
6.4 Continue Statement 138
6.5 Do Statement 139
6.6 Expression Statement 140
6.7 For Statement 141
6.8 Goto and Labeled Statements 143
6.9 If Statement 145
6.10 Null Statement 147
6.11 Return Statement 148
6.12 Switch Statement 150
6.13 While Statement 153

7 Functions 155

7.1 Introduction 157
7.2 Function Definitions 157
7.3 Function Declarations 165
7.4 Function Calls 167

8 Preprocessor Directives 175

8.1 Introduction 177
8.2 Manifest Constants and Macros 177
8.3 Include Files 182
8.4 Conditional Compilation 184
8.5 Line Control 188

Appendices 191

A Differences 193

B Syntax Summary 197

B.1 Tokens 199
B.2 Expressions 203
B.3 Declarations 205
B.4 Statements 208
B.5 Definitions 209
B.6 Preprocessor Directives 210

Index 211

Tables

Table 2.1 Punctuation and Special Characters 13
Table 2.2 Escape Sequences 14
Table 2.3 Operators 16
Table 2.4 Integer Constants 18
Table 2.5 Long Integer Constants 19
Table 2.6 Examples of Character Constants 21
Table 3.1 Summary of Lifetime and Visibility 35
Table 4.1 Fundamental Types 44
Table 4.2 Type Specifiers and Abbreviations 45
Table 4.3 Storage and Range of Values for Fundamental Types 46
Table 5.1 Precedence and Associativity of C Operators 120
Table 5.2 Conversions from Signed Integral Types 125
Table 5.3 Conversions from Unsigned Integral Types 127
Table 5.4 Conversions from Floating-Point Types 128

Chapter 1
Introduction

1.1 Overview 3
1.2 About This Manual 4
1.3 Notational Conventions 5

1.1 Overview

The C language is a general-purpose programming language well known for its efficiency, economy, and portability. While these advantages make it a good choice for almost any kind of programming, C has proven to be especially useful in systems programming because it allows programmers to write fast and compact programs and to transport those programs to other systems. In many cases, well-written C programs are comparable in speed to assembly language programs and offer the advantages of easier maintenance and greater readability.

In spite of C's efficiency and power, it is a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. Instead, C programmers rely on run-time libraries to perform such tasks.

This design contributes to C's adaptability and compactness. Because the language is relatively confined, it does not assume or impose a particular programming model. Run-time routines provide support as needed, allowing the programmer to minimize their use, if desired, or to tailor run-time routines for special purposes.

The design also helps to isolate language features from processor-specific features in a particular C implementation, thus aiding programmers who want to write portable code. The strict definition of the language makes it independent of any particular operating system or machine; at the same time, programmers can easily add system-specific routines to take advantage of a particular machine's efficiencies.

Some of the significant features of the C language are as follows.

- *C provides a full set of loop, conditional, and transfer statements* to control program flow logically and efficiently and to encourage structured programming.
- *C offers an unusually large set of operators.* Many of C's operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators lets the programmer specify different kinds of operations clearly and with a minimum of code.
- *C's data types include several sizes of integers,* as well as single- and double-precision floating-point types. The programmer can design more complex data types, such as arrays and data structures, to suit specific program needs.

- *C programmers can declare "pointers" to variables and functions.* A pointer to an item corresponds to the machine address of that item. Using pointers wisely can increase program efficiency considerably, since pointers let the programmer refer to items in the same way the machine does. C also supports pointer arithmetic, allowing the programmer both to access and manipulate memory addresses directly.
- *The C preprocessor, a text processor, acts on the text of files before compilation.* Among its most useful applications for C programs are the definition of program constants, the substitution of function calls with faster macro look-alikes, and conditional compilation. The preprocessor is not limited to processing C files; it can be used on any text file.
- *C is a flexible language,* leaving much of the decision-making up to the programmer. In keeping with this attitude, C imposes few restrictions in matters such as type conversion. While this is often an asset, it is important for C programmers to be thoroughly familiar with the definition of the language in order to understand how their programs will behave.

1.2 About This Manual

The *Microsoft® C Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers who have experience in C or in another programming language. Knowledge of programming fundamentals is assumed.

The run-time library functions available for use in Microsoft C programs are discussed in a separate library reference manual.

Consult your system documentation for an explanation of how to compile and link C programs on your system. Your system documentation also contains information specific to the implementation of C on your system.

This manual is organized as follows.

Chapter 2, "Elements of C," describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, "Program Structure," discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, "Declarations," describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and allows the programmer to declare aggregate types and pointers.

Chapter 5, "Expressions and Assignments," describes the operands and operators that make up C expressions and assignments. The type conversions and side effects that may accompany the evaluation of expressions are also discussed in this chapter.

Chapter 6, "Statements," describes C statements. Statements control the flow of program execution.

Chapter 7, "Functions," discusses features of C functions. In particular, it explains how to define, declare, and call a function and describes function parameters and return values.

Chapter 8, "Preprocessor Directives," describes the instructions recognized by the C preprocessor. The C preprocessor is a text processor automatically invoked before compilation.

Appendix A, "Differences," lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, "Syntax Summary," summarizes the syntax of the C language as implemented by Microsoft.

The remainder of this chapter describes the notational conventions used throughout the manual.

1.3 Notational Conventions

The following notational conventions are used throughout this manual.

italics	Italicized terms mark the places in syntax specifications and in the text where specific terms appear in an actual C program. For example, in **goto** *name*; *name* is italicized to show that this is a general

form for the **goto** statement. In an actual program statement, the user supplies a particular identifier for the placeholder *name*.

Italics are also used when referring to specific identifiers supplied for functions, variables, types, and labels. For example, when a program example such as

```
int *var[5];
```

is provided, the variable name *var* is italicized in the discussion of the example.

Occasionally italics are used to emphasize particular words in the text.

[brackets]

Brackets enclose optional items in syntax specifications. However, the C language also uses brackets for array declarations and subscript expressions. In discussions of the syntax for array declarations and subscript expressions, and in examples, brackets have the meaning specified by C. For instance,

```
return (var[1]);
```

is an example indicating that a function returns the value of the subscript expression *var*[1], whereas

return [*expression*];

is a syntax specification showing that *expression* is an optional item in the return statement.

ellipses...

Ellipses following an item indicate that more items having the same form may appear. Ellipses may be vertical or horizontal. For instance,

= {*expression* [, *expression*]...}

indicates that one or more expressions separated by commas may appear between the braces ({}). In the following example, the ellipses indicate that zero or more declarations, followed by

one or more statements, may appear between the braces.

```
{
        [declaration]
          .
          .
          .
        statement
        [statement]
          .
          .
          .
}
```

Vertical ellipses are also used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, two program lines are shown. The ellipses between the lines indicate that intervening program lines occur but are not shown.

```
int x, y;
   .
   .
   .
swap (&x, &y);
```

"quotation marks"

Quotation marks set off terms defined in the text. For example, the term "token" appears in quotation marks when it is defined.

Quotation marks are also used to set off program fragments and character values when discussing them in the text. For instance, a statement such as "x == z" appears in quotation marks. An escape sequence such as "\010", representing a character value, also appears in quotation marks in the text.

Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " rather than " ".

For example,

```
"abc"
```

is a C string.

keywords — C keywords, such as **goto** and **char**, are set in a different type font (the Helvetica font) to distinguish them from ordinary identifiers and text.

programming examples — Programming examples, such as the following, are displayed without proportional spacing to look similar to the programs created with a text editor.

```
int x, y;
   .
   .
   .
swap (&x, &y);
```

SMALL CAPITALS — Small capital letters are used for names of special key combinations such as CTRL-Z.

Chapter 2
Elements of C

2.1 Introduction 11
2.2 Character Sets 11
2.2.1 Letters and Digits 12
2.2.2 Whitespace Characters 12
2.2.3 Punctuation and Special Characters 13
2.2.4 Escape Sequences 14
2.2.5 Operators 15
2.3 Constants 17
2.3.1 Integer Constants 17
2.3.2 Floating-Point Constants 19
2.3.3 Character Constants 20
2.3.4 String Literals 21
2.4 Identifiers 23
2.5 Keywords 24
2.6 Comments 25
2.7 Tokens 26

2.1 Introduction

This chapter describes the elements of the C programming language. The elements of the language are the names, numbers, and characters used to construct a C program. In particular, this chapter describes

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens

2.2 Character Sets

Two character sets are defined for use in C programs, the C character set and the representable character set. The C character set consists of the letters, digits, and punctuation marks that have a specific meaning to the C compiler. C programs are constructed by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set consists of all letters, digits, and symbols that a user can represent graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

A C program can contain only characters from the C character set, except that string literals, character constants, and comments can use any representable character. Each character in the C character set has an explicit meaning to the C compiler. The compiler generates error messages when it encounters misused characters or characters not belonging to the C character set.

The following sections describe the characters and symbols of the C character set and explain how and when to use them.

2.2.1 Letters and Digits

The C character set includes the uppercase and lowercase letters of the English alphabet and the ten decimal digits of the Arabic number system:

Uppercase English letters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Lowercase English letters:
a b c d e f g h i j k l m n o p q r s t u v w x y z

Decimal digits:
0 1 2 3 4 5 6 7 8 9

These letters and digits can be used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. If a lowercase "a" is specified in a given item, you cannot substitute an uppercase "A" in its place; you must use the lowercase letter.

2.2.2 Whitespace Characters

Space, tab, linefeed, carriage return, form feed, vertical tab, and newline characters are called whitespace characters because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate user-defined items, such as constants and identifiers, from other items within a program.

A CTRL-Z character is treated as an end-of-file indicator. The compiler disregards any text following the CTRL-Z mark.

The C compiler ignores whitespace characters unless they are used as separators or as components of character constants or string literals. This means you can use extra whitespace characters to make a program more readable. Comments (see Section 2.6) are also treated as whitespace.

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set are used for a variety of purposes, from organizing the text of a program to defining the tasks to be carried out by the compiler or by the compiled program. Table 2.1 lists these characters.

Table 2.1

Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period	|	Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation	_	Underscore
"	Double quotation	#	Number sign
(	Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	-	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	+	Plus sign
>	Right angle bracket		

These characters have special meaning to the C compiler. Their use in the C language is described throughout this manual. Punctuation characters in the representable character set that do not appear in this list can be used only in string literals, character constants, and comments.

2.2.4 Escape Sequences

Escape sequences are special character combinations that represent white-space and nongraphic characters in strings and character constants. They are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of characters that normally have special meanings, such as the double quote (") character. An escape sequence consists of a backslash followed by a letter or combination of digits. Table 2.2 lists the C language escape sequences.

Table 2.2
Escape Sequences

Escape Sequence	Name
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash
ddd	ASCII character in octal notation
\x*dd*	ASCII character in hexadecimal notation

If the backslash precedes a character not included in the list above, the backslash is ignored and that character is represented literally. For example, the pattern "\c" represents the character "c" in a string literal or character constant.

The sequences "*ddd*" and "\x*dd*" allow any character in the ASCII character set to be given as a three-digit octal or a two-digit hexadecimal character code. For example, the backspace character can be given as "\010" or "\x08" The ASCII null character can be given as "\0" or "\x0".

Only octal digits can appear in an octal escape sequence, and at least one digit must appear. However, fewer than three digits can be specified. For example, the backspace character can also be given as "\10". Similarly, a hexadecimal escape sequence must contain at least one digit, but the second digit can be omitted. The hexadecimal escape sequence for the backspace character can be given as "\x8". However, when using octal and hexadecimal escape sequences in strings, it is safer to give all three digits of the octal or hexadecimal escape sequence. Otherwise, the character following the escape sequence may be interpreted as part of the sequence, if it happens to be an octal or hexadecimal digit.

Escape sequences allow nongraphic control characters to be sent to a display device. For example, the escape character, "\033", is often used as the first character of a control command for a terminal or printer.

Nongraphic characters should always be represented by escape sequences. Placing a nongraphic character in a C program has unpredictable results.

The backslash character (\) used to introduce escape sequences also functions as a continuation character in strings and in preprocessor definitions. When a newline character follows the backslash, the newline is disregarded, and the next line is treated as part of the previous line.

2.2.5 Operators

Operators are special character combinations that specify how values are to be transformed and assigned. The compiler interprets each of these character combinations as a single unit, called a "token" (see Section 2.7).

Table 2.3 lists the characters that form C operators and gives the name of each operator. Operators must be specified exactly as they appear in the tables, with no whitespace between the characters of multicharacter operators. The **sizeof** operator is not included in this table; it consists of a keyword (**sizeof**) rather than a symbol.

Table 2.3
Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
+	Addition
-	Subtraction, arithmetic negation
*	Multiplication, indirection
/	Division
%	Remainder
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equality
!=	Inequality
&	Bitwise AND, address-of
|	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR
,	Sequential evaluation
?:	Conditional[a]
++	Increment
--	Decrement
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
>>=	Right shift assignment
<<=	Left shift assignment
&=	Bitwise AND assignment
|=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

[a] The conditional operator is a ternary operator, not a multicharacter operator. The form of a conditional expression is
expression ? *expression* : *expression*

See Chapter 5, "Expressions and Assignments," for a complete description of each operator.

2.3 Constants

A constant is a number, a character, or a string of characters that can be used as a value in a program. The value of a constant does not change from execution to execution.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals. The following sections define the format and use of each.

2.3.1 Integer Constants

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value. A decimal constant has the form

digits

where *digits* are one or more decimal digits (0 through 9).

An octal constant has the form

0*odigits*

where *odigits* are one or more octal digits (0 through 7). The leading zero is required.

A hexadecimal constant has the form

0x*hdigits*

where *hdigits* is one or more hexadecimal digits (0 through 9 and either uppercase or lowercase "a" through "f"). The leading zero is required and must be followed by "x".

No whitespace characters can appear between the digits of an integer constant. Table 2.4 illustrates the form of integer constants.

Table 2.4
Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10	012	0xa or 0xA
132	0204	0x84
32179	076663	0x7dB3 or 0x7DB3

Integer constants always specify positive values. If negative values are required, the minus sign (-) can be placed in front of the constant to form a constant expression with a negative value. The minus sign is treated as an arithmetic operator.

Every integer constant is given a type based on its value. A constant's type determines what conversions must be performed when the constant is used in an expression or when the minus sign (-) is applied. Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.

Octal and hexadecimal constants are also given **int** type, or **long** type if the size of the value requires it. However, unlike other signed numbers, octal and hexadecimal constants are not sign-extended in type conversions.

The programmer can direct the C compiler to force any integer constant to have **long** type by appending the letter "l" or "L" to the end of the constant. Table 2.5 illustrates **long** integer constants.

Table 2.5
Long Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
`10L`	`012L`	`0xaL` or `0xAL`
`791`	`01151`	`0x4f1` or `0x4F1`

Types are described in Chapter 4, "Declarations," and conversions are described in Chapter 5, "Expressions and Assignments."

2.3.2 Floating-Point Constants

A floating-point constant is a decimal number representing a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. Floating-point constants have the form

[*digits*] [.*digits*] [E[−] *digits*]

where *digits* are one or more decimal digits (0 through 9), and E (or e) is the exponent symbol. Either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion) can be omitted, but not both. The exponent consists of the exponent symbol followed by a possibly negative constant integer value. The decimal point can be omitted only when an exponent is given. No whitespace characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. If negative values are required, the minus sign (−) can be placed in front of the constant to form a constant floating-point expression with a negative value. The minus sign is treated as an arithmetic operator.

The following examples illustrate some of the forms of floating-point constants and expressions.

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

The integer portion of the floating-point constant can be omitted, as shown in the following examples.

```
.75
.0075e2
-.125
-.175E-2
```

All floating-point constants have type **double**.

2.3.3 Character Constants

A character constant is a letter, digit, punctuation character, or escape sequence enclosed in single quotation marks. The value of a character constant is the character itself. Character constants consisting of more than one character or escape sequence are not allowed.

A character constant has the form

'char'

where *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a newline character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash as shown in Table 2.6. To represent a newline character, use the escape sequence '\n'.

Table 2.6
Examples of Character Constants

Constant	Value
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **char** and consequently are sign-extended in type conversions (see Section 5.7, "Type Conversions," of Chapter 5, "Expressions and Assignments").

2.3.4 String Literals

A string literal is a sequence of letters, digits, and symbols enclosed in double quotation marks. A string literal is treated as an array of characters; each element of the array is a single character value.

The form of a string literal is

"*characters*"

where *characters* are one or more characters from the representable character set, excluding the double quotation mark ("), the backslash (\), and the newline character. To use the newline character in a string, type a backslash immediately followed by a newline character. The backslash causes the newline character to be ignored. This allows the programmer

to form string literals that occupy more than one line. For example, the string literal

```
"Long strings can be bro\
ken into two pieces."
```

is identical to the string

```
"Long strings can be broken into two pieces."
```

To use the double quotation mark or backslash character within a string literal, precede it with a backslash, as shown in the following examples.

```
"This is a string literal."
"Enter a number between 1 and 100 \n Or press Return"
"First\\Second"
"\"Yes, I do,\" she said."
```

Notice that escape sequences (such as \n and \") can appear in string literals.

The characters of a string are stored in order at contiguous memory locations. A null character (\) is automatically appended to mark the end of the string. Each string in a program is considered to be a distinct item. If two identical strings appear in a program, they each receive distinct storage space.

String literals have the type **char []**. This means a string is an array whose elements have type **char**. The number of elements in the array is the number of characters in the string literal plus one, since the null character stored after the last character counts as an array element.

2.4 Identifiers

Identifiers are the names you supply for the variables, functions, and labels used in a given program. You can create an identifier by declaring it with the associated variable or function. You can use the identifier in later statements within the program to refer to the given item. (Declarations are described in Chapter 4, "Declarations.")

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Any number of characters are allowed in a given identifier, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may use fewer characters.) Use leading underscores with care. Identifiers beginning with an underscore can conflict with the names of hidden system routines and produce errors.

The following are examples of identifiers.

```
j
cnt
temp1
top_of_page
skip12
```

The C compiler considers uppercase and lowercase letters to be separate and distinct characters. Thus, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique.

```
add
ADD
Add
aDD
```

The C compiler does not allow identifiers that have the same spelling and case as a C language keyword. Keywords are described in Section 2.5.

The linker may further restrict the number and type of characters for globally visible symbols. Furthermore, unlike the compiler, the linker may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information on naming restrictions imposed by the linker.

2.5 Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. They can be used only as defined. The names of program items may not conflict with the keywords listed below.

auto	**default**	**float**	**register**	**switch**
break	**do**	**for**	**return**	**typedef**
case	**double**	**goto**	**short**	**union**
char	**else**	**if**	**sizeof**	**unsigned**
const	**enum**	**int**	**static**	**void**
continue	**extern**	**long**	**struct**	**while**

Keywords cannot be redefined. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see Chapter 8, "Preprocessor Directives").

The **const** keyword is reserved for future use but is not yet implemented in the language.

The following identifiers may be keywords in some implementations. See your system documentation for details.

far
fortran
huge
near
pascal

2.6 Comments

A comment is a sequence of characters that is treated as a single white-space character by the compiler but is otherwise ignored. A comment has the following form.

/ characters */*

Here *characters* can be any combination of characters from the representable character set, including newline characters but excluding the combination "*/". This means that comments can occupy more than one line, but they cannot be nested.

Comments are typically used to document the statements and actions of a C language source program. They can appear anywhere a whitespace character is allowed. Since the compiler ignores the characters of the comment, keywords can appear in comments without producing errors.

The following examples illustrate some comments.

```
/* Comments can separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/**************************************
   Comments can occupy several lines.
 **************************************/
```

Since comments cannot contain nested comments, the following example causes an error.

```
/* You cannot /* nest */ comments */
```

The compiler recognizes the first "*/", after the word "nest", as the end of the comment. The compiler attempts to process the remaining text and produces an error when it cannot do so.

To suppress compilation of a large portion of a program or a program segment that contains comments, use the **#if** preprocessor directive instead of comments (see Section 8.4, "Conditional Compilation," of Chapter 8, "Preprocessor Directives").

2.7 Tokens

When the compiler processes a program, it breaks the program down into groups of characters known as "tokens." A token is a unit of program text that has meaning to the compiler and that cannot be broken down further. The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets (**[]**), braces (**{}**), angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by whitespace characters and by other tokens, such as operators and punctuation symbols. To prevent the compiler from breaking an item down into two or more tokens, whitespace characters are prohibited between the characters of identifiers, multicharacter operators, and keywords.

When the compiler interprets tokens, it incorporates as many characters as possible into a single token before moving on to the next token. Because of this behavior, tokens not separated by whitespace may not be interpreted in the way expected.

For example, in the following expression, the compiler first makes the longest possible operator (++) from the three plus signs, and then processes the remaining plus sign as an addition operator (+).

```
i+++j
```

This expression is interpreted as "(i++) + (j)", not "(i) + (++j)". Use whitespace and parentheses to clarify your intent in such cases.

Chapter 3
Program Structure

3.1 Introduction 29
3.2 Source Program 29
3.3 Source Files 30
3.4 Program Execution 32
3.5 Lifetime and Visibility 33

3.1 Introduction

This chapter describes the structure of C language source programs and defines terms used later in this manual to describe the C language. It provides an overview of C language features that are described in detail in other chapters. In particular, the syntax and meaning of declarations and definitions are discussed in Chapter 4, "Declarations," and Chapter 7, "Functions." The C preprocessor is described in Chapter 8, "Preprocessor Directives."

3.2 Source Program

A C source program is a collection of one or more directives, declarations, and/or definitions. "Directives" instruct the C preprocessor to perform specific actions on the text of the program prior to compilation. "Declarations" establish the names and attributes of variables, functions, and types used in the program.

"Definitions" are declarations that also define variables and functions. A variable definition gives the initial value of the declared variable, in addition to its name and type. The definition causes storage to be allocated for the variable. A function definition specifies the function body, a compound statement containing the declarations and statements that constitute the function. The function definition also gives the function name, formal parameters, and return type.

A source program can have any number of directives, declarations, and definitions. Each must have the appropriate syntax as described in this manual. They can appear in any order in the program, although the order affects how variables and functions can be used in the program (see Section 3.5, "Lifetime and Visibility").

A nontrivial program always contains at least one definition, a function definition. The function defines the action to be taken by the program. The following example illustrates a simple C source program.

Example

```
int x = 1;                          /* Variable definitions */
int y = 2;

extern int printf(char *,);         /* Function declaration */

main ()                             /* Function definition
                                       for main function */
{
        int z;                      /* Variable declarations */
        int w;

        z = y + x;                  /* Executable statements */
        w = y - x;
        printf("z= %d \n w= %d \n", z, w);
}
```

This source program defines the function named *main* and declares the function *printf*. The variables *x* and *y* are defined with variable definitions; the variables *z* and *w* are just declared.

3.3 Source Files

Source programs can be divided into one or more source files. A C source file is a text file that contains all or part of a C source program. It may, for example, contain just a few of the functions needed by the program. When the source program is compiled, the individual source files that make up the program must be compiled individually and then linked. Separate source files can also be combined to form larger source files before compilation by using the **#include** directive, discussed in Chapter 8, "Preprocessor Directives."

A source file can contain any combination of complete directives, declarations, and definitions. Items such as function definitions or large data structures cannot be split between source files.

A source file need not contain any executable statements. It is sometimes useful to place variable definitions in one source file and then declare references to these variables in other source files that use them. This makes the definitions easy to find and modify if necessary. For the same reason, manifest constants and macros (discussed in Chapter 8, "Preprocessor Directives") are often organized into separate "include" files and inserted

into source files where required.

Directives in a source file apply to that source file and its included files only. Moreover, each directive applies only to the portion of the file following the directive. If a common set of directives is to be applied to a source program, then all source files in the program must contain these directives.

The following is an example of a C source program contained in two source files. The *main* and *max* functions are assumed to be in separate files, and execution of the program is assumed to begin with the *main* function.

Example

```
/**************************************************************
        Source file 1 - main function
 **************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int, int);       /* Function declaration */

main ()                         /* Function definition */
{
        int w = ONE, x = TWO, y = THREE;
        int z = 0;
        z = max(x,y);
        w = max(z,w);
}

/**************************************************************
        Source file 2 - max function
 **************************************************************/

int max (a, b)                  /* Function definition */
int a, b;
{
        if ( a > b )
                return (a);
        else
                return (b);
}
```

In the first source file, the function *max* is declared without being defined. This is known as a "forward declaration." The function definition for *main* includes function calls to *max*.

The lines beginning with a number sign (#) are preprocessor directives. These directives instruct the preprocessor to replace the identifiers ONE, TWO, and THREE with the specified number in the first source file. The directives do not apply to the second source file.

The second source file contains the function definition for *max*. This definition satisfies the calls to *max* in the first source file. Once the source files are compiled, they can be linked and executed as a single program.

3.4 Program Execution

Every program has a primary (main) program function. Traditionally, the primary program function is given the name "main". Many operating systems require this name for the primary function. The C language, however, does not explicitly define the name of the primary function.

This function serves as the starting point for program execution and usually controls execution of the program by directing the calls to other functions in the program. A program usually stops executing at the end of the main function, although it can stop at other points in the program, depending on the execution environment.

The source program usually has more than one function, each designed to perform one or more specific tasks. The main function can call these functions to perform the tasks. When a function is called, execution begins at the first statement in the called function. The function returns control when a **return** statement is executed or the end of the function is encountered.

All functions, including the main function, can be declared to have parameters. Functions called by other functions receive values for the parameters from the calling functions. Parameters of the main function can be declared to receive values passed to the main function from outside the program (for example, from the command line when the program is executed).

Traditionally, the first three parameters of the main function are declared to have the names "argc", "argv", and "envp". The "argc" parameter is declared to hold the total number of arguments passed to the main function. The "argv" parameter is declared as an array of pointers, each element of which points to a string representation of an argument passed to the main function. The "envp" parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the "argc", "argv", and "envp" parameters, and the user supplies the actual arguments to the main function. The argument-passing convention in use on a particular system is determined by the operating system rather than by the C language. See your system documentation for details.

Formal parameters to functions must be declared when the function is defined. Function definitions are described in more detail in Section 7.2 of Chapter 7, "Functions." Function declarations are discussed in Section 4.5 of Chapter 4, "Declarations."

3.5 Lifetime and Visibility

Two concepts, "lifetime" and "visibility," are important in understanding the structure of a C program. The lifetime of a variable or function can be either "global" or "local." An item with a global lifetime has storage and a defined value throughout the duration of the program. An item with a local lifetime is allocated new storage each time the "block" in which it is defined or declared is entered. When the block is exited, the local item loses its storage, and hence its value. Blocks are defined and discussed below.

An item is said to be "visible" in a block or source file if the type and name of the item are known in the block or source file. An item can also be "globally visible," which means that it is visible, or can be made visible through appropriate declarations, throughout all the source files that constitute the program. Visibility between source files (also known as "linkage") is discussed in greater detail in Section 4.6, "Storage Classes."

A block is a compound statement. Compound statements consist of declarations and statements, as described in Section 6.3 of Chapter 6, "Statements." The bodies of C functions are compound statements. Blocks can be nested; function bodies frequently contain blocks, which in turn can contain blocks.

Declarations and definitions within blocks are said to occur at the "internal level." Declarations and definitions outside of all blocks occur at the "external level."

Both variables and functions can be *declared* at the external level or at the internal level. Variables can also be *defined* at the internal level, but functions can only be defined at the external level.

All functions have global lifetimes, regardless of where they are declared. Variables declared at the external level always have global lifetimes. Variables declared at the internal level usually have local lifetimes; however, the storage class specifiers **static** and **extern** can be applied to declare global variables or references to global variables within a block. See Section 4.6, "Storage Classes," in Chapter 4, "Declarations," for a discussion of these options.

Variables declared or defined at the external level are visible from the point at which they are declared or defined to the end of the source file. These variables can be made visible in other source files with appropriate declarations, as described in Section 4.6, "Storage Classes," of Chapter 4, "Declarations." However, variables that are given **static** storage class at the external level are visible only within the source file in which they are defined.

In general, variables declared or defined at the internal level are visible from the point at which they are first declared or defined to the end of the block in which the definition or declaration appears. These variables are called local variables. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition supersedes the external level definition of the variable for the duration of the block. The visibility of the external level variable is restored when the block is exited.

Block visibility can nest. This means that a block nested inside another block can contain declarations that redefine variables declared in the outer block. The redefinition of the variable holds in the inner block, but the original definition is restored when control returns to the outer block. Variables from outer blocks are visible inside all inner blocks, as long as they are not redefined in the inner blocks.

Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. See Section 4.5 of Chapter 4, "Declarations," for more information on function declarations.

Table 3.1
Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable declaration	**static**	Global	Restricted to single source file
	Variable declaration	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

The following program example illustrates blocks, nesting, and visibility of variables.

Example

```
/* i defined at external level */
int i = 1;

/* main function defined at external level */
main ()
{

    /* prints 1 (value of external level i) */
    printf("%d\n", i);

    /* first nested block */
    {
    /* i and j defined at internal level */
    int i = 2, j = 3;

    /* prints 2, 3 */
    printf("%d\n%d\n", i, j);

    /* second nested block */
    {

        /* i is redefined */
        int i = 0;

        /* prints 0, 3 */
        printf("%d\n%d\n", i, j);

    /* end of second nested block */
    }

    /* prints 2 (outer definition restored) */
    printf("%d\n", i);

    /* end of first nested block */
    }

    /* prints 1 (external level definition restored) */
    printf("%d\n", i);
}
```

In this example, there are four levels of visibility: the external level and three block levels. Assuming that the function *printf* is defined elsewhere in the program, the *main* function prints out the values 1, 2, 3, 0, 3, 2, 1.

3.6 Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C allows you to use the same identifier for more than one program item, as long as you follow the rules outlined in this section.

The compiler sets up "naming classes" to distinguish between the identifiers for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in one or more naming classes. This means that you can use the same identifier for two or more different items if the items are in different naming classes. The context of a given identifier in the program allows the compiler to resolve the reference without ambiguity.

The kinds of items you can name in C programs, and the rules for naming them, are described below.

Variables and Functions	The names of variables and functions are in a naming class with formal parameters and enumeration constants. Variable and function names must, therefore, be distinct from other names in this class with the same visibility. However, variable names can be redefined within program blocks, as described in Section 3.5, "Lifetime and Visibility." Function names can also be redefined in this manner.
Formal Parameters	The names of formal parameters to a function are grouped with the names of the function's variables, so the formal parameter names should be distinct from the variable names. Redeclaring formal parameters within the function causes an error.

Enumeration Constants	Enumeration constants are in the same naming class as variable and function names. This means that names of enumeration constants must be distinct from all variable and function names with the same visibility and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, meaning that they can be redefined within blocks. See Section 3.5, "Lifetime and Visibility."
Tags	Enumeration, structure, and union tags are grouped together in a single naming class. Each enumeration, structure, or union tag must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from any other name in the program.
Statement Labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from any other names or from label names in other functions.
Typedef Names	The names of types defined with **typedef** are treated as keywords. No other names with the same visibility are allowed to have the same spelling and case as a **typedef** keyword.

Example

```
struct student {
        char student[20];
        int class;
        int id;
        } student;
```

Structure tags, structure members, and variable names are in three different naming classes, so no conflict occurs between the three items named *student* in the above example. The compiler determines how to interpret each occurrence of *student* by its context in the program. For example, when *student* appears after the **struct** keyword, it is known to be a structure tag. When *student* appears after either of the member selection operators "." or "->", the name refers to the structure member. In other contexts, the identifier *student* refers to the structure variable.

Chapter 4
Declarations

4.1 Introduction 43
4.2 Type Specifiers 44
4.3 Declarators 48
4.3.1 Pointer, Array, and Function Declarators 48
4.3.2 Complex Declarators 49
4.4 Variable Declarations 53
4.4.1 Simple Variable Declarations 54
4.4.2 Enumeration Declarations 55
4.4.3 Structure Declarations 57
4.4.4 Union Declarations 60
4.4.5 Array Declarations 62
4.4.6 Pointer Declarations 64
4.5 Function Declarations 66
4.6 Storage Classes 69
4.6.1 Variable Declarations at the External Level 70
4.6.2 Variable Declarations at the Internal Level 73
4.6.3 Function Declarations at the External and Internal Levels 75
4.7 Initialization 76
4.7.1 Fundamental and Pointer Types 77
4.7.2 Aggregate Types 77

4.7.3 String Initializers 79
4.8 Type Declarations 80
4.8.1 Structure, Union, and Enumeration Types 80
4.8.2 Typedef Declarations 81
4.9 Type Names 82

4.1 Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

[*sc-specifier*] [*type-specifier*] *declarator*[=*initializer*] [,*declarator...*]

where *sc-specifier* is a storage class specifier, *type-specifier* is the name of a defined type, *declarator* is an identifier that can be modified to declare a pointer, array or function, and *initializer* gives a value or sequence of values to be assigned to the variable being declared.

All C variables must be explicitly declared before they are used. C functions can be declared explicitly in a function declaration or implicitly by calling the function before it is declared or defined.

The C language defines a standard set of data types. You can add to that set by declaring new data types based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more *declarators*. A declarator is an identifier that can be modified with brackets (`[]`), asterisks (*), or parentheses to declare an array, pointer, or function type. When you declare simple variables (such as character, integer, and floating-point values), or structures and unions of simple variables, the declarator is just an identifier.

Four storage class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage class specifier of a declaration affects how the declared item is stored and initialized and which portions of a program can reference it. The location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Although function declarations are presented in Section 4.5 of this chapter, function definitions are described in Section 7.2 of Chapter 7, "Functions."

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called the "fundamental" types. Their names are listed in Table 4.1.

Table 4.1
Fundamental Types

Integral Types[a]	Floating-Point Types[a]	Other
char	**float**	**void**[b]
int	**double** (also called **long float**)	
short int		
long int		
unsigned char		
unsigned int		
unsigned short int		
unsigned long int		

[a] Used to declare variables and function return types.

[b] Used only to declare function return types.

Enumeration types are also considered fundamental types; type specifiers for enumeration types are discussed in Section 4.8.1. The **char**, **int**, **short int**, and **long int** types, together with their **unsigned** counterparts, are called "integral" types. The **float** and **double** type specifiers refer to "floating-point" types.

The **void** type can only be used to declare functions that return no value. Function types are discussed in Section 4.5, "Function Declarations."

You can create additional type specifiers with **typedef** declarations, discussed in Section 4.8.2.

Variable and function declarations can use any of the integral or floating-point type specifiers listed above. You can abbreviate some type specifiers, as shown in Table 4.2.

Table 4.2
Type Specifiers and Abbreviations

Type Specifier	Abbreviation
char	--
int	--
short int	short
long int	long
unsigned char	--
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long
float	--
long float	double

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type does not apply to variables, it is not included in the table.

Table 4.3

Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	−128 to 127
int	implementation-dependent	
short	2 bytes	−32,768 to 32,767
long	4 bytes	−2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation-dependent	
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE standard notation; see discussion below.
double	8 bytes	IEEE standard notation; see discussion below.

The **char** type is used to store a letter, digit, or symbol from the representable character set. The integer value of a character is the ASCII code corresponding to that character. Since the **char** type is interpreted as a signed 1-byte integer, values in the range −128 to 127 are permitted for **char** variables, although only the values from 0 to 127 have character equivalents.

Notice that the storage and range associated with the **int** and **unsigned int** types are not defined by the C language. Instead, the size of an **int** (signed or unsigned) corresponds to the natural size of an integer on a given machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent either to the **short int** or the **long int** type, depending on the implementation. Similarly, the **unsigned int** type is equivalent either to the **unsigned short** or **unsigned long** type.

The **int** and **unsigned int** type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the size of the **int** and **unsigned int** types varies, programs that depend on a specific **int** size may be nonportable. Expressions involving the **sizeof** operator (discussed in Section 5.3.4 of Chapter 5, "Expressions and Assignments") can be used in place of hard-coded data sizes to increase the portability of the code.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) are used to define certain features of the C language (for instance, in defining the **enum** type later in Section 4.8.1). In these cases, the definition of **int** and **unsigned int** for a particular implementation determines the actual storage.

The range of values for a variable lists the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in Chapter 5, "Expressions and Assignments"), it is not always possible to use the maximum or minimum for a variable of a given type in an expression.

For example, the constant-expression −32,768 consists of the arithmetic negation operator (−) applied to the constant value 32,768. Since 32,768 is too large to represent as a **short**, it is given **long** type, and the constant-expression −32,768 consequently has **long** type. The value −32,768 can only be represented as a **short** by type-casting it to the **short** type. No information is lost in the type cast, since −32,768 can be represented internally in 2 bytes of storage space.

Similarly, a value such as 65,000 can only be represented as an **unsigned short** by type-casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65,000 in decimal notation is considered a signed constant, and is given **long** type because 65,000 does not fit into a **short**. This **long** value can then be cast to the **unsigned short** type without loss of information, since 65,000 will fit into 2 bytes of storage space when it is stored as an unsigned number.

Octal and hexadecimal constants are considered unsigned quantities, even though they are given **int** or **long** type, because they have the special property of representing a bit pattern. Thus, octal and hexadecimal constants are not sign-extended in type conversions.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, a 7-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-

order bit of the mantissa is always 1, it is not stored in the number. This representation gives an exponent range of 10 to the (+ or −) 38th power and up to seven digits of precision. The maximum value of a **float** is normally 1.701411E38.

Values with **double** type have 8 bytes. The format is similar to the **float** format, except that the exponent is 11 bits excess 1023, and the mantissa has 52 bits (plus the implied high-order 1 bit). This gives an exponent range of 10 to the (+ or −) 306th power and up to 15 digits of precision.

4.3 Declarators

Syntax

identifier
declarator[]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

C allows the programmer to declare *arrays* of values, *pointers* to values, and *functions returning* values of specified types. To declare these items, you must use a *declarator*.

A declarator is an identifier possibly modified with brackets (`[]`), parentheses, and asterisks (*) to declare an array, pointer, or function type. Declarators appear in the pointer, array, and function declarations described in later sections of this chapter (Sections 4.4.6, 4.4.5, and 4.5, respectively). This section discusses the rules for forming and interpreting declarators.

4.3.1 Pointer, Array, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has an unmodified type. Asterisks (*) can appear to the left of an identifier, modifying it to a *pointer* type. If the identifier is followed by brackets (`[]`), the type is modified to an *array* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator does not constitute a complete declaration; a type specifier must be included as well. The type specifier gives the type of the elements for an array type, the type of object addressed by a pointer type, and the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail (see Sections 4.4.6, 4.4.5, and 4.5, respectively). The following examples illustrate the simplest forms of declarators.

Examples

```
1.  int list[20];

2.  char *cp;

3.  double func(void);
```

The above examples declare: 1. an array of **int** values (*list*); 2. a pointer to a **char** value (*cp*); and 3. a function with no arguments returning a **double** value (*func*).

4.3.2 Complex Declarators

Any declarator can be enclosed in parentheses. Parentheses are typically used to specify a particular interpretation of a "complex" declarator, as discussed below. A "complex" declarator is an identifier qualified by more than one array, pointer, or function modifier.

Various combinations of the array, pointer, and function modifiers can be applied to a single identifier. Some combinations are illegal. An array cannot be composed of functions, and a function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (on the right of the identifier) take precedence over asterisks (on the left of the identifier). Brackets and parentheses have the same precedence and associate left to right. The type specifier is applied as the last step, when the declarator has been fully interpreted. Parentheses can be used to override the default association order in a way that forces a particular interpretation.

A simple rule that can be helpful in interpreting complex declarators is to read them "from the inside out." Start with the identifier and look to the right for brackets or parentheses. Interpret these (if any), then look to the left for asterisks. If you encounter a right parenthesis at any stage, go back and apply these rules to everything within the parentheses before proceeding. As the last step, apply the type specifier. To illustrate this rule, the steps are numbered in order in the following example.

```
char *(*(*var)())[10];
 ^    ^ ^ ^ ^   ^   ^
 7    6 4 2 1   3   5
```

1. The identifier *var* is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

The following examples provide further illustration and show how parentheses can affect the meaning of a declaration.

Examples

```
1.    /* array of pointers to int values */
    int *var[5];

2.    /* pointer to array of int values */
    int (*var)[5];

3.    /* function returning pointer to long */
    long *var(long,long);

4.    /* pointer to function returning long */
    long (*var)(long,long);

5.    /* array of pointers to functions
         returning structures */
    struct both {
        int a;
        char b;
        } ( *var[ ] )( struct both, struct both );

6.    /* function returning pointer
         to an array of double values */
    double ( *var( double (*)[3] ) )[3];

7.    /* array of arrays of pointers
         to pointers to unions */
    union sign {
        int x;
        unsigned y;
        } **var[5][5];

8.    /* array of pointers to arrays
         of pointers to unions */
    union sign *(*var[5])[5];
```

In the first example, the array modifier has higher priority than the pointer modifier, so *var* is declared to be an array. The pointer modifier applies to the type of the array elements; the elements are pointers to **int** values.

In the second example, parentheses alter the meaning of the declaration in the first example. Now the pointer modifier has higher priority than the array modifier, and *var* is declared to be a pointer to an array of 5 **int** values.

Function modifiers also have higher priority than pointer modifiers, so the third example declares *var* to be a pointer to a function returning a **long** value. The function is declared to take two **long** values as arguments.

The fourth example is similar to the second example. Parentheses give the pointer modifier higher priority than the function modifier, and *var* is declared to be a pointer to a function returning a **long** value. Again, the function takes two **long** arguments.

The elements of an array may not be functions, but the fifth example demonstrates how to declare an array of pointers to functions instead. In this example *var* is declared to be an array of pointers to functions returning structures with two members. The arguments to the functions are declared to be two structures with the same structure type, *both*. Notice that the parentheses surrounding "*var[]" are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below.

```
                    /* ILLEGAL */
struct both *var[]( struct both, struct both );
```

The sixth example shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here *var* is declared to be a function returning a pointer to an array of 3 **double** values. The function *var* takes one argument; the argument, like the return value, is a pointer to an array of 3 **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of 3 pointers to **double** values. See Section 4.9, "Type Names", for a discussion and examples of abstract declarators.

A pointer can point to another pointer, and an array can contain array elements, as the seventh example shows. Here *var* is an array of 5 elements. Each element is a 5-element array of pointers to pointers to unions with two members.

The eighth example shows how the placement of parentheses alters the meaning of the declaration. In this example *var* is a 5-element array of pointers to 5-element arrays of pointers to unions.

4.4 Variable Declarations

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following.

Simple variables	Single value variables with integral or floating-point type.
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants.
Structures	Variables composed of a collection of values that may have different types.
Unions	Variables composed of several values of different types occupying the same storage space.
Arrays	Variables composed of a collection of elements with the same type.
Pointers	Variables that point to other variables. These variables contain variable locations (in the form of addresses) instead of values.

The variable declarations discussed in this section have the general form

[*sc-specifier*] *type-specifier declarator* [, *declarator*...]

where *type-specifier* gives the data type of the variable and *declarator* is the variable's name, possibly modified to declare an array or a pointer type. More than one variable can be defined in the declaration by giving multiple declarators, separated by commas.

The *sc-specifier* gives the storage class of the variable. In some contexts, variables can be initialized when they are declared. Storage classes and initialization are discussed in Sections 4.6 and 4.7, respectively.

4.4.1 Simple Variable Declarations

Syntax

sc-specifier type-specifier identifier [, *identifier...*] ;

A declaration for a simple variable defines the variable's name and type. It can also define the variable's storage class, as described later in Section 4.6. The variable's name is the *identifier* given in the declaration. The *type-specifier* gives the name of a defined data type, as described below.

You can define several variables in the same declaration by giving a list of identifiers separated by commas (,). Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Examples

```
1.  int x;

2.  unsigned long reply, flag;

3.  double order;
```

The first example defines a simple variable *x*. This variable can hold any value in the set defined by the **int** type in a particular implementation.

The second example defines two variables, *reply* and *flag*. Both variables have **unsigned long** type and hold unsigned integer values.

The third example defines a variable *order* that has **double** type. Floating-point values can be assigned to this variable.

4.4.2 Enumeration Declarations

Syntax

enum [*tag*] {*enum-list*} *identifier* [, *identifier...*];
enum *tag identifier* [, *identifier...*];

An enumeration declaration gives the name of the enumeration variable and defines a set of named integer constants (the "enumeration set"). A variable declared to have enumeration type stores any one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Enumeration declarations begin with the **enum** keyword and have two forms, as shown above. In the first form, the values and names of the enumeration set are specified in the *enum-list*, described in detail below. The optional *tag* is an identifier that names the enumeration type defined by the *enum-list*. The *identifier* names the enumeration variable. More than one enumeration variable can be defined in the declaration.

The second form uses an enumeration *tag* to refer to an enumeration type. The *enum-list* does not appear in this type of declaration because the enumeration type is defined elsewhere. An error is generated if the given *tag* does not refer to a defined enumeration type or if the named type is not currently visible.

An *enum-list* has the following form.

identifier [= *constant-expression*]
[, *identifier* [= *constant-expression*]]
.
.
.

Each *identifier* names a value of the enumeration set. By default, the first identifier is associated with the value zero, the next identifier is associated with the value one, and so on through the last identifier appearing in the declaration. The name of an enumeration constant is equivalent to its value.

The phrase "= *constant-expression*" overrides the default sequence of values. An identifier followed by the phrase "= *constant-expression*" is associated with the value given by *constant-expression*. The *constant-expression* must have **int** type and can be negative. The next identifier in the list is associated with the value of "*constant-expression* + 1", unless it is explicitly given another value.

An enumeration set can contain duplicate constant values, but each identifier in an enumeration list must be unique, that is, different from all other enumeration identifiers with the same visibility. For example, the value zero (0) could be given to two different identifiers, *null* and *zero*, in the same set. The identifiers in the list must also be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists. Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

```
1.  enum day {
        saturday,
        sunday = 0,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
        } workday;

2.  today = wednesday;

3.  enum day holiday;
```

The first example defines an enumeration type named *day* and declares a variable named *workday* with that enumeration type. The value 0 is associated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In the second example, a value from the set is assigned to the variable *today*. Notice that the name of the enumeration constant is used to assign the value.

In the third example, a variable named *holiday* is declared to have the enumeration type *day*. Since the *day* type was previously declared, only the enumeration tag is necessary in this declaration.

4.4.3 Structure Declarations

Syntax

struct [*tag*] {*member-declaration-list*} *declarator* [, *declarator...*];
struct *tag declarator* [, *declarator...*];

A structure declaration defines the name of the structure variable and specifies a sequence of variable values (called "members" of the structure) that can have different types. A variable with structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms, as shown above. In the first form, the types and names of the structure members are specified in the *member-declaration-list*, described in detail below. The optional *tag* is an identifier that names the structure type defined by the *member-declaration-list*.

Each *declarator* gives the name of a structure variable. The *declarator* may also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

The second form uses a structure *tag* to refer to a structure type. The *member-declaration-list* does not appear in this type of declaration because the structure type is defined elsewhere. The structure type definition must be visible for a *tag* declaration to be used, and the definition must appear prior to the *tag* declaration, unless the *tag* is used to declare a pointer variable or a **typedef** structure type. These declarations can use a structure *tag* before the structure type is defined, as long as the structure definition is visible to the declaration.

A *member-declaration-list* is a list of one or more variable or bitfield declarations. Each variable declared in the *member-declaration-list* is defined as a member of the structure type. Variable declarations within member declaration lists have the same form as the variable declarations discussed in this chapter, except that the declarations do not contain storage class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears. This allows you to create linked lists of structures.

Bitfields

A bitfield declaration has the following form.

type-specifier [*identifier*] : *constant-expression*;

The bitfield consists of the number of bits specified by *constant-expression*. The *type-specifier* for a bitfield declaration must specify an **unsigned** integral type, and the *constant-expression* must be a non-negative integer value. Arrays of bitfields, pointers to bitfields, and functions returning bitfields are not allowed. The optional *identifier* names the bitfield. An unnamed bitfield whose width is specified as zero (0) has a special function: it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

The identifiers in a structure declaration list must be unique within that list. It is not necessary for the identifiers in the list to be distinct from ordinary variable names or from identifiers in other structure declaration lists. Structure tags must be distinct from other structure, union, and enumeration tags having the same visibility.

Structure members are stored sequentially in the same order in which they are declared. The first member has the lowest memory address and the last member the highest. The storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can occur between the members of a structure in memory.

Bitfields are not stored across boundaries of their declared type. For example, a bitfield declared with **unsigned int** type is either packed into the space remaining in the previous **int** or it begins a new **int**.

Examples

```
1.  struct {
            float x,y;
    } complex;

2.  struct employee {
             char name[20];
             int id;
             long class;
    } temp;

3.  struct employee student, faculty, staff;

4.  struct sample {
             char c;
             float *pf;
             struct sample *next;
    } x;

5.  struct {
             unsigned icon : 8;
             unsigned color : 4;
             unsigned underline : 1;
             unsigned blink : 1;
    } screen[25][80];
```

The first example defines a structure variable named *complex*. This structure has two members with **float** type, *x* and *y*. The structure type is not named.

The second example defines a structure variable named *temp*. The structure has three members, *name*, *id*, and *class*. The *name* member is a 20-element array and *id* and *class* are simple members with **int** and **long** type, respectively. The identifier *employee* is the structure tag.

The third example defines three structure variables: *student*, *faculty*, and *staff*. Each structure has the same list of three members. The members are declared to have the structure type *employee*, defined in the previous example.

The fourth example defines a structure variable named *x*. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, *next*, is declared as a pointer to the structure type being defined (*sample*).

The fifth example defines a two-dimensional array of structures named *screen*. The array contains 2,000 elements. Each element is an individual structure containing four bitfield members: *icon*, *color*, *underline*, and *blink*.

4.4.4 Union Declarations

Syntax

union [*tag*] {*member-declaration-list*} *declarator* [, *declarator...*];
union *tag declarator*[, *declarator...*];

A union declaration defines the name of the union variable and specifies a set of variable values (called "members" of the union) that can have different types. A variable with union type stores any single value defined by that type.

Union declarations have the same forms as structure declarations except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bitfield members are not allowed in unions.

The storage associated with a union variable is the storage required for the longest member of the union. When a smaller member is used, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Examples

```
1.  union sign {
        int svar;
        unsigned uvar;
    } number;

2.  union {
        char *a, b;
        float f[20];
    } jack;

3.  union {
        struct {
                char icon;
                unsigned color : 4;
        } window1, window2, window3, window4;
    } screen[25][80];
```

The first example defines a union variable named *number* that has two members: *svar*, a signed integer, and *uvar*, an unsigned integer. This declaration allows the current value of *number* to be stored as either a signed or an unsigned value. The union type is named *sign*.

The second example defines a union variable named *jack*. The members of the union are, in order, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for *jack* is the storage required for the 20-element array *f*, since *f* is the longest member of the union. The union type is unnamed.

The third example defines a two-dimensional array of unions named *screen*. The array contains 2,000 elements. Each element is an individual union with four members: *window1*, *window2*, *window3*, and *window4*, where each member is a structure. Each union element holds one of the four possible structure members at any given time. Thus, the *screen* variable is a composite of up to four different "windows."

4.4.5 Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type-specifier declarator [];

A declaration for an array defines the name of the array and the type of each element. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in Section 5.2.2, "Identifiers," of Chapter 5, "Expressions and Assignments."

Array declarations have two forms, as shown above. The *declarator* gives the variable name, and may modify the variable's type. The brackets ([]) following the *declarator* modify the declarator to array type. The *constant-expression* inside the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier*. The *type-specifier* can specify any type except **void** and function types.

The second form omits the *constant-expression* in brackets. This form can be used only if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of arrays ("multidimensional" arrays) are defined by giving a list of bracketed *constant-expressions* following the array declarator.

type-specifier declarator[*constant-expression*] [*constant-expression*] ...

Each *constant-expression* in brackets defines the number of elements in a given dimension. Two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When a multidimensional array is declared within a function, the first *constant-expression* can be omitted if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of pointers to various types can be defined by using complex declarators, as described earlier in Section 4.3.2.

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks occur between the elements of an array in storage.

Arrays are stored by row. For example, the following array consists of 2 rows with 3 columns each.

```
char A[2][3];
```

The 3 columns of the first row are stored first, followed by the 3 columns of the second row.

To refer to an individual element of an array, use a subscript expression, discussed in Section 5.2.5 of Chapter 5, "Expressions and Assignments."

Examples

```
1.  int scores[10], game;

2.  float matrix[10][15];

3.  struct {
            float x,y;
            } complex[100];

4.  char *name[20];
```

The first example defines an array variable named *scores* with 10 elements, each of which has **int** type. The variable named *game* is declared as a simple variable with **int** type.

The second example defines a two-dimensional array named *matrix*. The array has 150 elements, each having **float** type.

The third example defines an array of structures. This array has 100 elements. Each element is a structure containing two members.

The fourth example defines an array of pointers. The array has 20 elements. Each element is a pointer to a **char** value.

4.4.6 Pointer Declarations

Syntax

type-specifier ***declarator**;

A pointer declaration defines the name of the pointer variable and the type of the object to which the variable points. The *declarator* defines the variable's name, and may modify its type. The *type-specifier* gives the type of the object. The type can be any fundamental, structure, or union type.

Pointer variables can also point to functions, arrays, and other pointers. To declare more complex pointer types, refer to Section 4.3.2, "Complex Declarators."

A pointer to a structure or union type can be declared before the structure or union type is defined, as long as the structure or union type definition is visible at the time of the declaration. Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable. The pointer is declared by using the structure or union tag. See the fourth example below.

A variable declared as a pointer holds a memory address. The amount of storage required for an address and the meaning of the address depends on the given implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations the special keywords **near** and **far** are available to modify the size of a pointer. See your system documentation for more information.

Examples

```
1.  char *message;

2.  int *pointers[10];

3.  int (*pointer)[10];

4.  struct list *next, *previous;

5.  struct list {
            char *token;
            int count;
            struct list *next;
    } line;
```

The first example defines a pointer variable named *message*. It points to a variable with **char** type.

The second example defines an array of pointers named *pointers*. The array has 10 elements. Each element is a pointer to a variable with **int** type.

The third example defines a pointer variable named *pointer*. It points to an array with 10 elements. Each element in this array has **int** type.

The fourth example defines two pointer variables that point to the structure type *list*. This declaration can appear before the definition of the *list* structure type (see the next example), as long as the *list* type definition has the same visibility as the declaration.

The fifth example declares the variable *line* to have the structure type named *list*. The *list* structure type is defined to have three members. The first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another *list* structure.

4.5 Function Declarations

Syntax

[*type-specifier*] *declarator*([*arg-type-list*]) [, *declarator...*] ;

A function declaration defines the name and return type of a function, and possibly establishes the types and number of arguments to the function. Function declarations, also called forward declarations, do not define the function body or parameters. Instead they permit the attributes of the function to be known before the function is defined. Function definitions are described in detail in Section 7.2 of Chapter 7, "Functions."

The *declarator* of the function declaration names the function, and the *type-specifier* gives the function's return type. If the *type-specifier* is omitted from a function declaration, the return type of the function is assumed to be **int**.

Function declarations may include either the **extern** or the **static** storage class specifier. Storage class specifiers are discussed in Section 4.6.

Argument Type List

The *arg-type-list* establishes the number and types of the arguments to the function. It has the following form.

[*type-name*] [, *type-name...*] [,]

The first *type-name* gives the type of the first argument to the function, the second *type-name* gives the type of the second argument, and so on. Each *type-name* is separated from the next by a comma. If the *arg-type-list* ends with a comma, the number of arguments to the function is variable. However, the function is expected to have at least as many arguments as there are *type-names* before the terminating comma. If the *arg-type-list* contains only a single comma, the number of arguments to the function is variable and may be zero.

A *type-name* for a fundamental, structure, or union type consists of the type specifier for that type (such as **int**). The *type-names* for pointers, arrays, and functions are formed by combining a type specifier with an "abstract declarator," that is, a declarator without an identifier. Section 4.9, "Type Names," explains how to form and interpret abstract declarators.

The special keyword **void** can be used in place of the *arg-type-list* to declare a function that has no arguments. The compiler produces a warning message if a call to the function or the function definition specifies arguments.

One other special construction is allowed in the *arg-type-list*. The phrase **void** * specifies an argument of any pointer type. This phrase can be used in the *arg-type-list* as if it were a *type-name*.

The *arg-type-list* may be omitted altogether. The parentheses after the function identifier are still required, but they are empty. In this form the function declaration establishes neither the number nor the types of arguments to the function. When this information is omitted, the compiler does not perform any type-checking between the actual arguments in a function call and the formal parameters of the function definition. See Section 7.4, "Function Calls," in Chapter 7, "Functions," for details.

Return Type

Functions can return values of any type except arrays and functions. Thus, the *type-specifier* of a function declaration can specify any fundamental, structure, or union type. The function identifier can be modified with one or more asterisks (*) to declare a pointer return type.

Although functions are not allowed to return arrays and functions, they can return pointers to arrays and functions. Functions that return pointers to array or function types are declared by modifying the function identifier with asterisks (*), brackets ([]), and parentheses to form a complex declarator. Forming and interpreting complex declarators is discussed in Section 4.3.2.

Examples

```
1.  int add(int, int);

2.  char *strfind(char *,);

3.  void draw(void);

4.  double (*sum(double, double))[3];

5.  int (*(*select)(void))(int)

6.  char *p;
    short *q;
    int prt(void *);
```

The first example declares a function named *add* that takes two **int** arguments and returns an **int** value.

The second example declares a function named *strfind*, which returns a pointer to a **char** value. The function takes at least one argument, a pointer to a **char** value. The argument type list ends with a comma, indicating that the function may take more arguments.

The third example declares a function with **void** return type (returning no value). The *argument-type-list* is also **void**, meaning no arguments are expected for this function.

In the fourth example, *sum* is declared as a function returning a pointer to an array of 3 **double** values. The *sum* function takes two arguments, each a **double** value.

In the fifth example, the function named *select* is declared to return a pointer to a function taking no arguments and returning a pointer. The pointer return value points to a function taking one **int** argument and returning an **int** value.

In the sixth example, the function *prt* is declared to take a pointer argument of any type and to return an **int**. Either the **char** pointer *p* or the **short** pointer *q* could be passed as an argument to *prt* without producing a type mismatch warning.

4.6 Storage Classes

The storage class of a variable determines whether the item has a "global" or "local" lifetime. An item with a global lifetime exists and has a value throughout the duration of the program. All functions have global lifetimes.

Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution passes out of the block, the variables no longer have meaningful values.

Although C defines only two types of storage classes, four storage class specifiers are available. They are

auto
register
static
extern

Items with **auto** and **register** class have local lifetimes. The **static** and **extern** specifiers refer to items with global lifetimes.

The four storage class specifiers have distinct meanings because storage class specifiers affect the visibility of functions and variables as well as their storage class. The term "visibility" refers to the portion of the source program in which the variable or function can be referenced. An item with a global lifetime exists throughout the execution of the source program, but it may not be "visible" in all parts of the program. Visibility and the related concept of lifetime are discussed in Section 3.5 of Chapter 3, "Program Structure."

The placement of variable or function declarations within source files also affects storage class and visibility. Declarations outside of all function definitions are said to occur at the "external level"; declarations within function definitions occur at the "internal level."

The exact meaning of each storage class specifier depends on whether the declaration occurs at the external or the internal level and whether the item declared is a variable or a function. The following sections describe the meaning of storage class specifiers in each kind of declaration. They also explain the default behavior when the storage class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the External Level

Variable declarations at the external level use the **static** and **extern** storage class specifiers or omit the storage class specifier entirely. The **auto** and **register** storage class specifiers are not allowed at the external level.

Variable declarations at the external level are either *definitions* of variables or *references* to variables defined elsewhere. An external variable declaration that also initializes the variable (implicitly or explicitly) is a definition of the variable. Definitions at the external level can take several forms:

1. A variable can be defined at the external level by declaring it with the **static** storage class specifier. The **static** variable can be explicitly initialized, as described in Section 4.7. If the initializer is omitted, the variable is automatically initialized to zero at compile time. Thus, "static int k = 16;" and "static int k;" are both considered definitions.
2. A variable is defined when it is explicitly initialized at the external level. For example, "int j = 3;" is a variable definition.

Once a variable is defined at the external level, it is visible throughout the remainder of the source file in which it appears. The variable is not visible above its definition in the same source file, nor is it visible in other source files of the program, unless a *reference* is declared to make it visible, as described below.

A variable can be defined at the external level only once within a source file. If the **static** storage class specifier is given, another variable with the same name can be defined with the **static** storage class specifier in a different source file. Since each **static** definition is visible only in its own source file, no conflict occurs.

The **extern** storage class specifier is used to declare a *reference* to a variable defined elsewhere. These declarations can be used to make a definition in another source file visible or to make a variable visible above its definition in the same source file. Once a reference to the variable is declared at the external level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage class specifier are not allowed to contain initializers, since they refer to variables whose values are already defined.

For an **extern** reference to be valid, the variable to which it refers must be defined once, and only once, at the external level. The definition can be in any of the source files that make up the program.

One special case is not covered by the rules outlined above. You can omit both the storage class specifier and the initializer from a variable declaration at the external level. For example, the declaration "int n;" is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

1. If a variable by the same name is *defined* at the external level elsewhere in the program, the declaration is taken to be a reference to that variable, exactly as if the **extern** storage class specifier had been used in the declaration.
2. If no such definition is present, the declared variable is allocated storage at link time and initialized to zero. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of *i* at the external level, "int i;" and "char i;", storage space for an **int** is allocated for *i* at link time.

Example

```
*************************************************************
                    SOURCE FILE ONE
*************************************************************

extern int i;                       /* reference to i,
                                       defined below */

main()
{
        i++;
        printf("%d\n", i);    /* i equals 4 */
        next();
}

int i = 3;                          /* definition of i */

next()
{
        i++;
        printf("%d\n", i);    /* i equals 5 */
        other();
}

*************************************************************
                    SOURCE FILE TWO
*************************************************************

extern int i;                       /* reference to i in
                                        first source file */

other()
{
        i++;
        printf("%d\n", i);    /* i equals 6 */
}
```

The two source files contain a total of three external declarations of *i*. Only one declaration contains an initialization: that declaration, "int i = 3;", defines the global variable *i* with initial value 3. The **extern** declaration of *i* at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the *main* function could not reference the global variable *i*. The **extern** declaration of *i* in the second source file makes the global variable visible in that source file.

All three functions perform the same task: they increase *i* and print it. (Assume that the *printf* function is defined elsewhere in the program.) The values printed are 4, 5, and 6.

If the variable *i* had not been initialized, it would have been automatically set to zero at link time. The values printed in this case would be 1, 2, and 3.

4.6.2 Variable Declarations at the Internal Level

Any of the four storage class specifiers can be used for variable declarations at the internal level. When the storage class specifier is omitted from a variable declaration at the internal level, the default storage class is **auto**.

The **auto** storage class specifier declares a variable with a local lifetime. The variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed later in this chapter. Variables with **auto** storage class are not initialized automatically, so they should be explicitly initialized when declared or assigned initial values in statements within the block. If not initialized, the values of **auto** variables are undefined.

The **register** storage class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually results in faster access time and smaller code size. Variables declared with **register** storage class have the same visibility as **auto** variables.

The number of registers that can be used for variable storage is machine dependent. If no registers are available when the compiler encounters the **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in exactly the same order in which the declarations appear in the source file. Register storage (if available) is only guaranteed for **int** and pointer types.

A variable declared at the internal level with the **static** storage class specifier has a global lifetime. The variable is visible only within the block in which it is declared. Unlike **auto** variables, variables declared as **static** retain their values when the block is exited.

Declarations of **static** variables can include initializers. If not explicitly initialized, a **static** variable is automatically set to zero. Initialization is performed once, at compile time; the **static** variable is *not* reinitialized each time the block is entered.

A variable declared with the **extern** storage class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The purpose of the internal **extern** declaration is to make the external-level variable definition visible within the block. The internal **extern** declaration does not change the visibility of the global variable in any other part of the program.

Example

```
int i = 1;

main()
{       /* reference to i, defined above */
        extern int i;

        /* initial value is zero; a is
           visible only within main */
        static int a;

        /* b is stored in a register, if possible */
        register int b = 0;

        /* default storage class is auto */
        int c = 0;

        /* values printed are 1, 0, 0, 0 */
        printf("%d\n%d\n%d\n%d\n", i, a, b, c);
        other();
}

other()
{
        /* i is redefined */
        int i = 16;

        /* this a is visible only within other */
        static int a = 2;

        a += 2;
        /* values printed are 16, 4 */
        printf("%d\n%d\n", i, a);
}
```

The variable *i* is defined at the external level with initial value 1. A reference to the external-level *i* is declared in the *main* function with an **extern** declaration. The **static** variable *a* is automatically set to zero, since the initializer is omitted. The call to *printf* (assuming the *printf* function is defined elsewhere in the source program) prints out the values 1, 0, 0, 0.

In the *other* function, the variable *i* is redefined as a local variable with initial value 16. This does not affect the value of the external-level *i*. The variable *a* is declared as a **static** variable and initialized to 2. This *a* does not conflict with the *a* in *main,* since the visibility of **static** variables at the internal level is restricted to the block in which they are declared.

The variable *a* is increased by 2, giving 4 as the result. If the *other* function were called again in the same program, the initial value of *a* would be 4. Internal **static** variables retain their values when the block in which they are declared is exited and reentered.

4.6.3 Function Declarations at the External and Internal Levels

Function declarations can use either the **static** or the **extern** storage class specifier. Functions always have global lifetimes.

The visibility rules for functions are slightly different from the rules for variables. Function declarations at the internal level have the same meaning as function declarations at the external level. This means that functions cannot have block visibility, and the visibility of functions cannot be nested. A function declared to be **static** is visible only within the source file in which it is defined. Any function in the same source file can call the **static** function, but functions in other source files cannot. Another **static** function by the same name can be declared in a different source file without conflict.

Functions declared as **extern** are visible throughout all the source files that constitute the program (unless they are later redeclared as **static**). Any function can call an **extern** function.

Function declarations that omit the storage class specifier default to **extern**.

4.7 Initialization

A variable can be set to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. The initializer is preceded by an equal sign (=), as shown below.

= *initializer*

Variables of any type can be initialized, with the restrictions outlined below. Functions do not take initializers.

Declarations that use the **extern** storage class specifier cannot contain initializers.

Variables declared at the external level can be initialized; if not explicitly initialized, they are set to zero at compile time. Any variable declared with the **static** storage class specifier can be initialized. Initializations of **static** variables are performed once, at compile time. If not explicitly initialized, **static** variables are automatically set to zero.

Initializations of **auto** and **register** variables are performed each time execution control passes to the block in which they are declared. If the initializer is omitted from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.

Initializations of **auto** aggregate types (arrays, structures, and unions) are prohibited. Only **static** aggregates and aggregates declared at the external level can be initialized.

The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant-expressions. Constant-expressions are described in Section 5.2.10 of Chapter 5, "Expressions and Assignments." Automatic and register variables can be initialized with either constant or variable values.

Sections 4.7.1 and 4.7.2 describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

Examples

```
1.  int x = 10;

2.  register int *px = 0;

3.  int c = (3 * 1024);

4.  int *b = &x;
```

In the first example, *x* is initialized to the constant-expression 10. In the second example, the pointer *px* is initialized to zero, producing a "null" pointer. The third example uses a constant-expression to initialize *c*. The fourth example initializes the pointer *b* with the address of another variable, *x*.

4.7.2 Aggregate Types

Syntax

= {*initializer-list*}

An *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant-expression or an *initializer-list*. Thus, a brace-enclosed list can appear within another *initializer-list*. This is useful for initializing aggregate members of an aggregate, as shown in the examples below.

For each *initializer-list*, the values of the constant-expressions are assigned in order to the members of the aggregate variable. When a union is initialized, the *initializer-list* must be a single constant-expression. The value of the constant-expression is assigned to the first member of the union.

If there are fewer values in an *initializer-list* than there are in the aggregate type, the remaining members or elements are initialized to zero. Giving too many initial values for the aggregate type causes an error. These rules apply to each embedded *initializer-list*, as well as to the aggregate as a whole.

For example,

```
int P[4][3] = {
        { 1, 1, 1 },
        { 2, 2, 2 },
        { 3, 3, 3,},
        { 4, 4, 4,},
};
```

declares P as a 4 x 3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Notice that the *initializer-list* for the third and fourth rows contain commas after the last constant-expression. The last *initializer-list* ("{4, 4, 4,}") is also followed by a comma. These extra commas are permitted but are not required; the required commas are those that separate constant-expressions and *initializer-lists*.

If there is no embedded initializer list for an aggregate member, values are simply assigned in order to each member of the subaggregate. Thus, the above initialization is equivalent to

```
int P[4][3] = {
        1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

Examples

```
1.  struct list {
        int i, j, k;
        float m[2][3];
        } x = {
                1,
                2,
                3,
                {4.0, 4.0, 4.0}
        };

2.  union {
        char x[2][3];
        int i, j, k;
        } y = {
                {'1'},
                {'4'}
        };
```

In the first example, the three **int** members of *x* are initialized to 1, 2, and 3, respectively. The three elements in the first row of *m* are initialized to 4.0; the elements of the remaining row of *m* are initialized to zero by default.

In the second example the union variable *y* is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list "**{'1'}**" gives values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character "1", and the remaining two elements in the row are initialized to zero (the null character), by default. Similarly, the first element of the second row of "x" is initialized to the character "4", and the remaining two elements in the row are initialized to zero.

4.7.3 String Initializers

An array can be initialized with a string literal. For example,

```
char code[ ] = "abc";
```

initializes *code* as a four-element array of characters. The fourth element is the null character that terminates all string literals.

If the array size is specified and the string is longer than the specified size of the array, the extra characters are simply discarded. The following declaration initializes *code* as a three-element character array.

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to *code*. The character "d" and the null character are discarded.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to zero (the null character).

4.8 Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. The name of a declared type can be used in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. These declarations are used to set up shorter or more meaningful names for types already defined by C or for types declared by the user.

4.8.1 Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. In type declarations the variable identifier is omitted, since no variable is declared. The *tag* is mandatory; it names the structure, union, or enumeration type. The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which a *tag* refers to a type defined elsewhere, is not legal for type declarations.

Examples

```
1.  enum status {
        loss = -1,
        bye,
        tie = 0,
        win
        };

2.  struct student {
        char name[20];
        int id, class;
        };
```

The first example declares an enumeration type named *status*. The name of the type can be used in declarations of enumeration variables. The identifier *loss* is explicitly set to −1. Both *bye* and *tie* are associated with the value 0, and *win* is given the value 1.

The second example declares a structure type named *student*. A structure variable can be declared to have *student* type with a declaration such as "struct student employee;".

4.8.2 Typedef Declarations

Syntax

typedef *type-specifier declarator* [, *declarator...*] ;

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword appears in place of a storage class specifier. The declaration is interpreted in the same way as variable and function declarations, but the identifier, instead of taking on the type specified by the declaration, becomes a new keyword for the type.

Typedef does not create types. It creates synonyms for existing types or names for types that could be specified in other ways. Any type can be declared with **typedef**, including pointer, function, and array types. A **typedef** name for a pointer to a structure or union type can be declared before the structure or union type is defined, as long as the definition has the same visibility as the declaration.

Examples

```
1.  typedef int WHOLE;

2.  typedef struct club {
        char name[30];
        int size, year;
        } GROUP ;

3.  typedef GROUP *PG;

4.  typedef void DRAWF(int, int);
```

The first example declares *WHOLE* to be a synonym for **int**.

The second example declares *GROUP* as a structure type with three members. Since a structure tag, *club*, is also specified, either the **typedef** name (*GROUP*) or the structure tag can be used in declarations.

The third example uses the previous **typedef** name to declare a pointer type. The type *PG* is declared as a pointer to the *GROUP* type, which in turn is defined as a structure type.

The final example provides the type *DRAWF* for a function returning no value and taking two **int** arguments. This means, for example, that the declaration "DRAWF box;" is equivalent to the declaration "void box(int, int);"

4.9 Type Names

A "type name" specifies a particular data type. Type names are used in three contexts: in the argument type lists of function declarations, in type casts, and in **sizeof** operations. Argument type lists are discussed in Section 4.5, "Function Declarations." Type casts and **sizeof** operations are discussed in Section 5.7.2 and 5.3.4, respectively, of Chapter 5, "Expressions and Assignments."

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the form

type-specifier abstract-declarator

An *abstract declarator* is a declarator without an identifier, consisting solely of one or more pointer, array, or function modifiers. The pointer modifier (*) always appears before the identifier in a declarator, while array (`[]`) and function (()) modifiers appear after the identifier. It is thus possible to determine where the identifier would appear in an abstract declarator and to interpret the declarator accordingly.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

When you give a function type with an abstract declarator, you can include the function's argument type list, which also consists of type names. See the second and fourth examples below.

The abstract declarator "()" alone is not allowed because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (a function type).

The type specifiers established through **typedef** declarations also qualify as type names.

Examples

```
1.  long *

2.  double *(double, double)

3.  int (*)[5]

4.  int (*)(void)
```

The first example gives the type name for "pointer to **long**" type.

The second example is the type name for a function that takes two **double** arguments and returns a pointer to a **double** value.

The third and fourth examples show how parentheses modify complex abstract declarators. Example 3 gives the name for a pointer to an array of five **int** values. Example 4 names a pointer to a function taking no arguments and returning an **int**.

Chapter 5
Expressions and Assignments

5.1 Introduction 87
5.2 Operands 87
5.2.1 Constants 88
5.2.2 Identifiers 88
5.2.3 Strings 89
5.2.4 Function Calls 89
5.2.5 Subscript Expressions 90
5.2.6 Member Selection Expressions 92
5.2.7 Expressions with Operators 94
5.2.8 Expressions in Parentheses 95
5.2.9 Type-Cast Expressions 95
5.2.10 Constant-Expressions 95
5.3 Operators 97
5.3.1 Usual Arithmetic Conversions 97
5.3.2 Complement Operators 99
5.3.3 Indirection and Address-of Operators 100
5.3.4 Sizeof Operator 101
5.3.5 Multiplicative Operators 102
5.3.6 Additive Operators 104
5.3.7 Shift Operators 107
5.3.8 Relational Operators 108

5.3.9 Bitwise Operators 110
5.3.10 Logical Operators 112
5.3.11 Sequential Evaluation Operator 114
5.3.12 Conditional Operator 115
5.4 Assignment Operators 116
5.4.1 Lvalue Expressions 116
5.4.2 Unary Increment and Decrement 117
5.4.3 Simple Assignment 118
5.4.4 Compound Assignment 118
5.5 Precedence and Order of Evaluation 119
5.6 Side Effects 123
5.7 Type Conversions 124
5.7.1 Assignment Conversions 124
5.7.1.1 Conversions from Signed Integral Types 124
5.7.1.2 Conversions from Unsigned Integral Types 126
5.7.1.3 Conversions from Floating-Point Types 128
5.7.1.4 Conversions from Other Types 129
5.7.2 Type-Cast Conversions 129
5.7.3 Operator Conversions 130
5.7.4 Function-Call Conversions 130

5.1 Introduction

This chapter describes how to form expressions and make assignments in the C language. An expression is a combination of operands and operators that yields ("expresses") a single value. An operand is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. Operators specify how the operand or operands of the expression are manipulated.

In C, assignments are considered expressions. An assignment yields a value. Its value is the value being assigned. In addition to the simple assignment operator (=), C offers complex assignment operators that both transform and assign their operands.

The value resulting from an expression's evaluation depends on the relative precedence of operators in the expression and side effects, if present. The precedence of operators determines the grouping of operands in an expression. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression.

The value represented by each operand in an expression has a type, which may be converted to a different type in certain contexts. Type conversions take place in assignments, type casts, function calls, and operations.

5.2 Operands

A C operand is a constant, an identifier, a string, a function call, a subscript expression, a member selection expression, or a more complex expression formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a "constant-expression."

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be cast from its original type to another type by means of a "type-cast" operation. A type-cast expression can also form an operand of an expression.

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **char** type. An integer constant can have either **int** or **long** type, depending on the integer's size and how the value was specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed in Section 5.2.3.

5.2.2 Identifiers

An identifier names a variable or function. Every identifier has a type, which is established when the identifier is declared. The value of an identifier depends upon its type, as follows.

- Identifiers of integral and floating-point types represent values of the corresponding type.
- An identifier of **enum** type represents one constant value of a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.
- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.
- An identifier declared as a pointer represents a pointer to the specified type.
- An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer is the type of the first element of the array. For example, if *series* is declared to be a ten-element integer array, the identifier *series* expresses the address of the array, while the subscript expression "series[*n*]" (where *n* is an integer in the range zero to nine) refers to a variable integer element of *series*. Subscript expressions are discussed in Section 5.2.5.

  The address of an array does not change during the execution of the program, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, and an array identifier cannot form the left-hand operand of an assignment operation.

- An identifier declared as a function represents a pointer whose value is the address of the function. The type addressed by the pointer is a function returning a value of a specified type. The address of a function does not change during the execution of a program; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

A string literal consists of a list of characters enclosed in double quotes, as shown below.

"*string*"

A string literal is stored as an array of elements with **char** type. The string literal represents the address of the first element of the array. The address of the string's first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in contexts that allow pointer values, and they are subject to the same restrictions as pointers. String literals have one additional restriction: they are not variables and cannot be left-hand operands in assignment operations.

The last character of a string is always the null character, "\0". The null character is not visible in the string expression, but it is added as the last element when the string is stored. Thus, the string "abc" actually has four characters rather than three.

5.2.4 Function Calls

Syntax

expression (*expression-list*)

A function call consists of an *expression* followed by an *expression-list* in parentheses, where *expression* evaluates to a function address (for example, a function identifier), and *expression-list* is a list of expressions whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

A function call expression has the value and type of the function's return value. If the function's return type is **void**, the function call expression also has **void** type. If control returns from the called function without execution of a **return** statement, the value of the function call is undefined.

See Section 7.4 of Chapter 7, "Functions," for a detailed discussion of function calls.

5.2.5 Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. *Expression1* is any pointer value (such as an array identifier) and *expression2* is an integral value. *Expression2* must be enclosed in brackets ([]).

Subscript expressions are generally used to refer to array elements, but a subscript can be applied to any pointer.

The subscript expression is evaluated by adding the integral value (*expression2*) to the pointer value (*expression1*), then applying the indirection operator (*) to the result. (See Section 5.3.3 for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that *a* is a pointer and *b* an integer.

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules of the addition operator (see Section 5.3.6) the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier *line* refers to an array of **int** values. To evaluate the expression *line*[*i*], the integer value *i* is multiplied by the length of an **int**. The converted value of *i* represents *i* **int** positions. This converted value is added to the original pointer value (*line*) to yield an address that is offset *i* **int** positions from *line*.

As the last step in evaluating the subscript expression, the indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, *line*[*i*]).

Notice that the subscript expression

```
line[0]
```

represents the value of the first element of A, since the offset from the address represented by A is zero. Similarly, an expression such as

```
line[5]
```

refers to the element offset five positions from *line*, or the sixth element of the array.

Multidimensional Array References

A subscript expression can be subscripted, as follows.

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate left to right. The leftmost subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms the pointer expression to which *expression3* is added. The indirection operator (*) is applied after the last subscripted expression is evaluated. However, the indirection operator is not applied at all if the final pointer value addresses an array type. See the third example below.

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. The first element of a three-dimensional array, for example, is an array with two dimensions.

Examples

```
int prop[3][4][6];
int i, *ip;

1.  i = prop[0][0][1];

2.  i = prop[2][1][3];

3.  ip = prop[2][1];
```

The array named *prop* has 3 elements, each of which is a 4-by-6 array of int values.

Example 1 shows how to refer to the second individual int element of *prop*. Arrays are stored by row, so the last subscript varies fastest.

The second example shows a more complex reference to an individual element of *prop*. To evaluate the expression, the first subscript, 2, is multiplied by the size of a 4-by-6 int array and added to the pointer value *prop*. The result points to a 6-element array, the third element of the selected 4-by-6 array.

Next the second subscript, 1, is multiplied by the size of the 6-element int array and added to the address represented by *prop* [2].

Each element of the 6-element array is an int value, so the final subscript, 3, is multiplied by the size of an int before it is added to *prop* [2] [1]. The resulting pointer addresses the fourth element of the 6-element array.

The last step in evaluating the expression *prop* [2] [1] [3] is applying the indirection operator to the pointer value. The result is the int element at that address.

Example 3 shows a case where the indirection operator is not applied. The expression *prop* [2] [1] is a valid reference to the 3-dimensional array *prop*; the result of the expression is a pointer value that addresses an array with 1 dimension. Since the pointer value addresses an array type, the indirection operator is not applied.

5.2.6 Member Selection Expressions

Syntax

expression.identifier
expression->identifier

Member selection expressions refer to members of structures and unions. A member selection expression has the value and type of the selected member.

In the first form, "*expression.identifier*", *expression* represents a value of **struct** or **union** type. The *identifier* names a member of the specified structure or union.

In the second form, "*expression->identifier*", *expression* represents a pointer to a structure or union. The *identifier* names a member of the specified structure or union.

The two forms of member selection expressions have a similar effect. In fact, expressions involving the pointer selection operator (->) are shorthand versions of expressions using the period (.) in cases where the expression before the period consists of the indirection operator (*) applied to a pointer value. (The indirection operator is discussed in Section 5.3.3.) Thus,

expression->identifier

is equivalent to

(**expression*).*identifier*

when *expression* is a pointer value.

Examples

```
struct pair {
        int a;
        int b;
        struct pair *sp;
        } item, list[10];

1.  item.sp = &item;

2.  (item.sp)->a = 24;

3.  list[8].b = 12;
```

In the first example, the address of the *item* structure is assigned to the *sp* member of the structure. This means that *item* contains a pointer to itself.

In the second example, the pointer expression "item.sp" is used with the pointer selection operator (->) to assign a value to the member *a*.

The third example shows how to select an individual structure member from an array of structures.

5.2.7 Expressions with Operators

Expressions with operators can be unary, binary, or ternary expressions. A unary expression consists of an operand prefixed by an unary operator ("unop") or an operand enclosed in parentheses and preceded by the **sizeof** keyword:

unop operand
sizeof (*operand*)

A binary expression consists of two operands joined by a binary operator ("binop"):

operand binop operand

A ternary expression consists of three operands joined by the ternary (? :) operator:

operand ? *operand* : *operand*

Assignment expressions use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators. The binary assignment operators are the simple assignment operator (=) and the compound assignment operators (referred to as "compound-assign-ops"). Each compound assignment operator is a combination of another binary operator with the simple assignment operator. The forms of assignment expressions are

operand++
operand--
++*operand*
--*operand*
operand = *operand*
operand compound-assignment-op operand

5.2.8 Expressions in Parentheses

Any operand can be enclosed in parentheses. The parentheses have no effect on the type or value of the enclosed expression. For example, in the expression

```
(10 + 5) / 5
```

the parentheses around "10 + 5" mean that the value of "10 + 5" is the left operand of the "/" (division) operator. The result of "(10 + 5) / 5" is 3. Without the parentheses, "10 + 5 / 5" would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation for the expression.

5.2.9 Type-Cast Expressions

A type-cast expression has the following form.

(*type-name*) *operand*

Type-cast conversions are discussed in Section 5.7.2; type names are discussed in Section 4.9 of Chapter 4, "Declarations."

5.2.10 Constant-Expressions

A constant-expression is any expression that evaluates to a constant. The operands of a constant-expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant-expressions. The operands can be combined and modified using operators, as described in Section 5.2.7, with some restrictions.

Constant-expressions may not use assignment operators (see Section 5.4) or the binary sequential evaluation operator (,). The unary address-of operator (&) can be used only in certain initializations (see the last paragraph of Section 5.2.10).

Constant-expressions used in preprocessor directives are subject to additional restrictions, and are consequently known as *restricted-constant-expressions*. A *restricted-constant-expression* cannot contain **sizeof** expressions, enumeration constants, or type casts to any type. It can, however,

contain the special constant-expression "defined(*identifier*)". See Section 8.2.1 of Chapter 8, "Preprocessor Directives," for details.

These additional restrictions also apply to constant-expressions used to initialize variables at the external level. However, such expressions are allowed to apply the unary address-of operator (&) to other external-level variables with fundamental, structure, and union types and to external-level arrays subscripted with a constant-expression. In these expressions, a constant-expression not involving the address-of operator can be added to or subtracted from the address expression.

5.3 Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator).

Unary operators prefix their operand and associate right to left. C's unary operators are

- ~ !	Complement operators
* &	Indirection and address-of operators
sizeof	Size operator

Binary operators associate left to right. The binary operators are

* / %	Multiplicative operators
+ -	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential evaluation operator

C has one ternary operator, the conditional operator (? :). It associates right to left.

5.3.1 Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating-point types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are called the "usual arithmetic conversions." The discussion of each operator in the following sections specifies whether the operator performs the usual arithmetic conversions and also specifies the additional conversions, if any, the operator performs.

The specific path of each type of conversion is outlined in Section 5.7, "Type Conversions."

The usual arithmetic conversions proceed in order as follows.

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **double** type, the other operand is converted to **double**.
3. Any operands of **char** or **short** type are converted to **int**.
4. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
5. If one operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. If one operand is of type **long**, the other operand is converted to **long**.
7. If one operand is of type **unsigned int**, the other operand is converted to **unsigned int**.

5.3.2 Complement Operators

Arithmetic Negation (-)

The arithmetic negation operator (-) produces the negative (two's complement) of its operand. The operand must be an integral or floating-point value. The usual arithmetic conversions are performed.

Bitwise Complement (~)

The bitwise complement operator (~) produces the bitwise complement of its operand. The operand must be of integral type. The usual arithmetic conversions are performed. The result has the type of the operand after conversion.

Logical NOT (!)

The logical NOT operator (!) produces the value zero if its operand is true (nonzero) and the value one if its operand is false (zero). The result has int type. The operand must be an integral, floating-point, or pointer value.

Examples

```
1.  short x = 987;
    x = -x;

2.  unsigned short y = 0xaaaa;
    y = ~y;

3.  if ( !(x < y));
```

In the first example, the new value of x is the negative of 987, or -987.

In the second example, the new value assigned to y is the one's complement of the unsigned value 0xaaaa, or 0x5555.

In the third example, if x is greater than or equal to y, the result of the expression is one (true). If x is less than y, the result is zero (false).

5.3.3 Indirection and Address-of Operators

Indirection (*)

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value to which the operand points. The result type is the type addressed by the pointer operand. If the pointer value is null, the result is unpredictable.

Address-of (&)

The address-of operator (&) takes the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. (Assignment operations are discussed in Section 5.4.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator cannot be applied to a bitfield member of a structure, nor can it be applied to an identifier declared with the **register** storage class specifier.

Examples

```
int *pa, x;
int a[20];

1.   pa = &a[5];

2.   x = *pa;
```

In the first example, the address-of operator (&) takes the address of the sixth element of the array *a*. The result is stored in the pointer variable *pa*.

The indirection operator (*) is used in the second example to access the int value at the address stored in *pa*. The value is assigned to the integer variable *x*.

5.3.4 Sizeof Operator

The **sizeof** operator determines the amount of storage associated with an identifier or a type. A **sizeof** expression has the form

sizeof(*name*)

where *name* is either an identifier or a type name. The type name may not be **void**. The value of a **sizeof** expression is the amount of storage, in bytes, associated with the named identifier or type.

When the **sizeof** operator is applied to an array identifier, the result is the size of the entire array in bytes rather than the size of the pointer represented by the array identifier.

When the **sizeof** operator is applied to a structure or union type name, or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union, which may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the members.

Example

```
buffer = calloc(100, sizeof (int) );
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The above example uses the **sizeof** operator to pass the size of an **int**, which varies across machines, as an argument to a function named *calloc*. The value returned by the function is stored in *buffer*.

5.3.5 Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral; the multiplication (*) and division (/) operators take integral and floating-point operands. The types of the operands can be different. The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

The conversions performed by the multiplicative operators make no provision for overflow or underflow conditions. Information is lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*)

The multiplication operator (*) specifies that its two operands are to be multiplied.

Division (/)

The division operator (/) specifies that its first operand is to be divided by the second. When two integers are divided, the result, if not an integer, is truncated. If both operands are positive or unsigned, the result is truncated toward zero. The direction of truncation when either operand is negative may be either toward or away from zero, depending on the implementation. Division by zero gives unpredictable results.

Remainder (%)

The result of the remainder operator (%) is the remainder when the first operand is divided by the second.

Examples

```
int i = 10, j = 3, n;
double x = 2.0, y;

1.  y = x * i;

2.  n = i / j;

3.  n = i % j;
```

In the first example, x is multiplied by i to give the value 20.0. The result has **double** type.

In the second example, 10 is divided by 3. The result is truncated toward zero, yielding the integer value 3.

In the third example, n is assigned the integer remainder 1 when 10 is divided by 3.

5.3.6 Additive Operators

The additive operators perform addition (+) and subtraction (−). The operands can be integral or floating-point values; some additive operations can also be performed on pointer values, as outlined under the discussion of each operator. The usual arithmetic conversions are performed on integral and floating-point operands. The type of the result is the type of the operands after conversion.

The conversions performed by the additive operators make no provision for overflow or underflow conditions. Information is lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) specifies addition of its two operands. The operands can have integral or floating-point types, as described above, or one operand can be a pointer and the other an integer. When an integer is added to a pointer, the integer value (i) is converted by multiplying it by the length of the value addressed by the pointer. After conversion, the integer value represents i memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value expressing the address i positions from the original address. The new pointer value addresses the same type as the original pointer value.

Subtraction (−)

The subtraction operator (−) subtracts its second operand from the first. The operands can be integral or floating-point values, as described above. The subtraction operator also allows the subtraction of an integer from a pointer value and the subtraction of two pointer values.

When an integer value is subtracted from a pointer value, the same conversions take place as with addition of a pointer and integer. The subtraction operator converts the integer value with respect to the type addressed by the pointer value. The result is the memory address i positions before the original address, where i is the integer value and each position is the length of the type addressed by the pointer value. The new pointer points to the type addressed by the original pointer value.

Two pointer values can be subtracted if they point to the same type. The difference between the two pointers is converted to a signed integer value by dividing the difference by the length of the type the pointers address. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed below.

Pointer Arithmetic

Additive operations involving a pointer and an integer generally give meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only memory positions of the same size lie between the original address and the address plus offset.

This assumption is valid for array members. An array is by definition a series of values of the same type; its elements reside in contiguous memory locations. Storage of any types except array elements is not guaranteed to be completely filled. That is, blanks can occur between memory positions, even positions of the same type. Adding to or subtracting from addresses referring to any values but array elements gives unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that only values of the same type, with no blanks, lie between the two addresses given by the operands.

Additive operations between pointer and integer values on machines with segmented architecture must take the segment addressing conventions into account. In some cases these operations may not be valid. See your system documentation for more information.

Examples

```
int i = 4, j;
float x[10];
float *px;

1.   px = &x[4] + i;

2.   j = &x[i] - &x[i-2];
```

In the first example, the integer operand *i* is added to the address of the

fifth element of *x*. The value of *i* is multiplied by the length of a **float** and added to "&x[4]" . The resulting pointer value is the address of "x[8]"

In the second example, the address of the third element of *x* ("x[i-2]") is subtracted from the address of the fifth element of *x* ("x[i]"). The difference is divided by the length of a **float**. The result is the integer value -2.

5.3.7 Shift Operators

The shift operators shift their first operand left ($<<$) or right ($>>$) by the number of positions the second operand specifies. Both operands must be integral values. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

For leftward shifts, the vacated right bits are filled with zeros. In a rightward shift, the method of filling left bits depends on the type (after conversion) of the first operand. If it is **unsigned**, vacated left bits will be filled with zeros. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

The conversions performed by the shift operators make no provision for overflow or underflow conditions. Information is lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In the above example, x is shifted left by 8 positions and y is shifted right 8 positions. The shifted values are added, giving 0xaa55, and assigned to z.

5.3.8 Relational Operators

The binary relational operators test their first operand against the second to determine if the relation specified by the operator holds true. The result of a relational expression is either one (if the tested relation holds) or zero (if it does not). The type of the result is **int**. The relational operators test the following relationships.

<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The operands can have integral, floating-point, or pointer type. The types of the operands can be different. The usual arithmetic conversions are performed on integral and floating-point operands.

One or both operands of the equality (==) and inequality (!=) operators can have **enum** type. An **enum** value is converted in the same manner as an **int** value.

The operands of any relational operator can be two pointers to the same type. For the equality (==) and inequality (!=) operators the result of the comparison reflects whether the two pointers address the same memory location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects the relative position of two memory addresses.

Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. Comparisons between the addresses of different elements of the same array can be useful, however, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is "less than" the address of the last element.

A pointer value can be compared for equality (==) or inequality (!=) to the constant value zero (0). A pointer with a value of zero does not point to a memory location: it is called a "null" pointer. A pointer value is equal to zero only if it is explicitly given that value through assignment or initialization.

Examples

```
int x = 0, y = 0;
1.  x < y
2.  x > y
3.  x <= y
4.  x >= y
5.  x == y
6.  x != y
```

When x and y are equal, expressions 3, 4, and 5 have the value one and expressions 1, 2, and 6 have the value zero.

5.3.9 Bitwise Operators

The bitwise operators perform bitwise AND (&), inclusive OR (|), and exclusive OR (^) operations. The operands of bitwise operators must have integral type, but their types can be different. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise AND (&) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are ones, the corresponding bit of the result is set to one. Otherwise, the corresponding result bit is set to zero.

Bitwise Inclusive OR (|)

The bitwise inclusive OR (|) operator compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is a one, the corresponding bit of the result is set to one. Otherwise, both bits are zeros, and the corresponding result bit is set to zero.

Bitwise Exclusive OR (^)

The bitwise exclusive OR (^) operator compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is a zero and the other bit is a one, the corresponding bit of the

result is set to one. Otherwise, the corresponding result bit is set to zero.

Examples

```
short i = 0xab00;
short j = 0xabcd;
short n;

1.   n = i & j;

2.   n = i | j;

3.   n = i ^ j;
```

The result assigned to *n* in the first example is the same as *i*, 0xab00. The bitwise inclusive OR in the second example results in the value 0xabcd, while the bitwise exclusive OR in the third example produces 0x00cd.

5.3.10 Logical Operators

The logical operators perform logical AND (&&) and OR (| |) operations. The operands of the logical operators must have integral, floating-point, or pointer type. The types of the operands can be different.

The operands of logical AND and OR expressions are evaluated left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated.

These operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to zero. A pointer has a value of zero only if it is explicitly set to zero through assignment or initialization.

The result of a logical operation is either zero or one, as described below. The type of the result is **int**.

Logical AND (&&)

The logical AND operator (&&) produces the value one if both operands have nonzero values. If either operand is equal to zero, the result is zero. If the first operand of a logical AND operation has a value of zero, the second operand is not evaluated.

Logical OR (| |)

The logical OR operator (| |) performs an inclusive OR on its operands. It produces the value zero if both operands have zero values. If either operand has a nonzero value, the result is one. If the first operand of a logical OR operation has a nonzero value, the second operand is not evaluated.

Examples

```
int x, y;

1.  if (x < y && y < z)
        printf ("x is less than z\n");

2.  if (x == y || x == z)
        printf ("x is equal to either y or z\n");
```

In the first example, the *printf* function is called to print a message if x is less than y and y is less than z. If x is greater than y, "y < z" is not evaluated and nothing is printed.

In the second example, a message is printed if x is equal to either y or z. If x is equal to y, "x == z" is not evaluated.

5.3.11 Sequential Evaluation Operator

The sequential evaluation operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The types of the operands are unrestricted. No conversions are performed.

This operator (also called the "comma" operator) is typically used to evaluate two or more expressions in contexts that allow only one expression to appear.

Examples

```
1.  for ( i = j = 1; i + j < 20; i += i, j--);

2.  f(x, y + 2, z);
    f((x--, y + 2), z);
```

In the first example, each operand of the **for** statement's third expression is evaluated independently. The left operand, "i += i," is evaluated first, then "j--" is evaluated.

As shown in the second example, the comma character is used in other contexts as a separator. In the first function call, three arguments, separated by commas, are passed to the called function: *x*, "y + 2", and *z*. The use of the comma character as a separator must not be confused with its use as an operator; the two functions are completely different.

In the second function call, parentheses force the compiler to interpret the first comma as the sequential evaluation operator. This function call passes two arguments to *f*. The first argument is the result of the sequential evaluation operation "(x--, y + 2)", which has the value and type of the expression "y + 2"; the second argument is *z*.

5.3.12 Conditional Operator

C has one ternary operator, the conditional operator (? :). Its form is

operand1 ? *operand2* : *operand3*

Operand1 is evaluated in terms of its equivalence to zero. It must have integral, floating-point, or pointer type. If *operand1* has a nonzero value, *operand2* is evaluated and the result of the expression is the value of *operand2*. If *operand1* evaluates to zero, *operand3* is evaluated, and the result of the expression is the value of *operand3*. Notice that either *operand2* or *operand3* is evaluated, but not both.

The type of the result depends on the types of the second and third operands, as follows.

1. If both the second and third operands have integral or floating-point type (their types can be different), the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.
2. Both the second and third operands can have the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.
3. One of the second or third operands can be a pointer and the other a constant-expression with the value zero. The type of the result is the pointer type.

Example

```
j = (i < 0) ? (-i) : (i);
```

The above example assigns the absolute value of i to j. If i is less than zero, $-i$ is assigned to j. If i is greater than or equal to zero, i is assigned to j.

5.4 Assignment Operators

C's assignment operators can both transform and assign values in a single operation. Using a compound assignment operator to replace two separate operations can reduce code size and improve program efficiency. The assignment operators are listed and described below.

++	Unary increment operator
--	Unary decrement operator
=	Simple assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator
+=	Addition assignment operator
-=	Subtraction assignment operator
<<=	Left shift assignment operator
>>=	Right shift assignment operator
&=	Bitwise AND assignment operator
|=	Bitwise inclusive OR assignment operator
^=	Bitwise exclusive OR assignment operator

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific path of the conversion depends on the two types and is outlined in detail in Section 5.7, "Type Conversions."

5.4.1 Lvalue Expressions

An assignment operation specifies that the value of the right-hand operand is to be assigned to the storage location named by the left-hand operand. Thus, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression referring to a memory location. Expressions that refer to memory locations are called "lvalue" expressions. A variable name is such an expression: the name of the variable denotes a storage location, while the value of the variable is the value residing at that location.

The C expressions that may be lvalue expressions are

- identifiers of character, integer, floating-point, pointer, enumeration, structure, or union type
- subscript ([]) expressions, except when a subscript expression evaluates to a pointer to an array
- member selection expressions (-> and .), if the selected member is one of the above expressions
- unary indirection (*) expressions, except when such expressions refer to arrays
- type casts to pointer types
- an lvalue expression in parentheses

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and --) increase and decrease their operand, respectively. The operand must have integral, floating-point, or pointer type, and must be an lvalue expression.

Operands of integral or floating-point type are increased or decreased by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is increased or decreased by the size of the object it addresses. An increased pointer points to the next object; a decreased pointer points to the previous object.

An increment (++) or decrement (--) operator can appear either before or after its operand. When the operator prefixes its operand, the result of the expression is the increased or decreased value of the operand. When the operator postfixes its operand, the immediate result of the expression is the value of the operand *before* it is increased or decreased. After that result is noted in context, the operand is increased or decreased.

Examples

```
1.  if (pos++ > 0)
        *ptt = *qtt;

2.  if (line[--i] != '\n')
        return;
```

In the first example, the variable *pos* is compared to zero, then increased by one.

In the second example, the variable *i* is decreased before it is used as a subscript to *line*.

5.4.3 Simple Assignment

The simple assignment operator (=) performs assignment. The right operand is assigned to the left operand; the conversion rules for assignment (discussed in Section 5.7.1) apply.

Example

```
double x;
int y;

x = y;
```

The value of *y* is converted to **double** type and assigned to *x*.

5.4.4 Compound Assignment

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. A compound assignment expression such as

expression1 += *expression2*

can be understood as

expression1 = *expression1* + *expression2*

However, the compound assignment expression is not equivalent to the expanded version because the compound assignment expression evaluates *expression1* only once, while in the expanded version *expression1* is evaluated twice: in the addition operation and in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs, and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example

```
#define MASK     0xffff

n |= MASK;
```

In this example a bitwise inclusive OR operation is performed on *n* and MASK , and the result is assigned to *n*. The manifest constant MASK is defined with a **#define** preprocessor directive, discussed in Section 8.2.1 of Chapter 8, "Preprocessor Directives."

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in an expression. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions involving higher precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators. The operators are listed in order of precedence from the highest to the lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity, that is, either left to right or right to left.

Table 5.1
Precedence and Associativity of C Operators

Operator[a] Symbol	Type of Operation	Associativity
( ) [ ] . ->	Expression	Left to right
- ~ ! * & ++ -- **sizeof** *casts*	Unary[b]	Right to left
* / %	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Shift	Left to right
< > <= >=	Relational (inequality)	Left to right
== !=	Relational (equality)	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
= *= /= %= += -= <<= >>= &= |= ^=	Simple and compound assignment[c]	Right to left
,	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. Where several operators appear in the same line or in a large brace, they have equal precedence.

[b] All unary operators have equal precedence.

[c] All simple and compound assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member selection expression, or a parenthetical expression have highest precedence and associate left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either right to left or left to right. The result of expressions involving multiple occurrences of multiplication (*), addition (+), or binary bitwise (&, |, ^) operators at the same level is indifferent to the direction of evaluation. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

Only the sequential evaluation operator (,) and the logical AND (&&) and OR (| |) operators guarantee a particular order of evaluation for the operands. The sequential evaluation operator (,) is guaranteed to evaluate its operands from left to right.

The logical operators also guarantee to evaluate their operands left to right. However, the logical operators evaluate the minimum number of operands necessary to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression "x && y++" , the second operand, "y++" , is evaluated only if *x* is true (nonzero). Thus, *y* is not increased when *x* is false (zero).

The examples below show the default grouping for several expressions.

Examples

Expression	Default Grouping
1. `a & b || c`	`(a & b) || c`
2. `a = b || c`	`a = (b || c)`
3. `q && r || s--`	`(q && r) || s--`

In the first example, the bitwise AND operator (&) has higher precedence than the logical OR operator (| |), so "a & b" forms the first operand of the logical OR operation.

In the second example, the logical OR operator (| |) has higher precedence than the simple assignment operator (=), so "b | | c" is grouped as the right-hand operand in the assignment. Notice that the value assigned to *a* is either zero or one.

The third example shows a correctly formed expression that may produce an unexpected result. The logical AND operator (&&) has higher precedence than the logical OR operator (| |), so "q && r" is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, "q && r" is evaluated before "s--". However, if "q && r" evaluates to a nonzero value, "s--" is not evaluated, and "s" is not decreased. To correct this problem, "s--" should appear as the first operand of the expression or should be decreased in a separate operation.

The following example shows an illegal expression that produces a program error.

Example

Illegal Expression	Default Grouping
`p == 0 ? p += 1: p += 2`	`(p == 0 ? p += 1 : p) += 2`

In this example, the equality operator (==) has the highest precedence, so "p == 0" is grouped as an operand. The ternary operator (? :) has the next highest precedence. Its first operand is "p ==0" and its second operand is "p += 1". However, the last operand of the ternary operator is considered to be "p" rather than "p += 2", since this occurrence of *p* binds more closely to the ternary operator than it does to the compound assignment operator. A syntax error occurs because "+= 2" does not have a left-hand operand.

To prevent errors of this kind, and to produce more readable code, the use of parentheses is recommended. The above example can be corrected and clarified through the use of parentheses, as shown here.

```
(p == 0) ? (p += 1) : (p += 2)
```

5.6 Side Effects

"Side effects" are changes in the state of the machine that take place as a result of evaluating an expression. Side effects occur whenever the value of a variable is changed. Any assignment operation has side effects, and any call to a function that contains assignment operations has side effects.

The order of evaluation of side effects is implementation-dependent, except where the compiler guarantees a particular order of evaluation, as outlined in Section 5.5.

For example, side effects occur in the following function call.

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression "i + 1" may be evaluated before "i = j + 2", or vice versa, with different results in each case.

Unary increment and decrement operations involve assignment and can cause side effects, as shown in the following example.

```
d = 0;
a = b++ = c++ = d++;
```

The value of *a* is unpredictable. The initial value of *d* (zero) could be assigned to *c*, then to *b*, and then to *a* before any of the variables are increased. In this case *a* would be equal to zero.

A second method of evaluating this expression begins by evaluating the operand "c++ = d++". The initial value of *d* (zero) is assigned to *c*, and then both *d* and *c* are increased. Next, the increased value of *c* (one) is assigned to *b* and *b* is increased. Finally, the increased value of *b* is assigned to *a*. In this case, the final value of *a* is two.

Since the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be implemented. Statements that depend on a particular order of evaluation for side effects produce nonportable and unclear code.

5.7 Type Conversions

Type conversions take place when a value is assigned to a variable of a different type, when a value is explicitly cast to another type, when an operator converts the type of its operand or operands before performing an operation, and when a value is passed as an argument to a function. The rules governing each kind of conversion are outlined below.

5.7.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integral and floating-point types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows.

5.7.1.1 Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits and is converted to a longer signed integer by sign-extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a **long** value is converted to a **float**. To convert a signed integer to an unsigned integer, the signed integer is converted to the size of the unsigned integer and the result is interpreted as an unsigned value.

Conversions from signed integral types are summarized in Table 5.2.

Table 5.2
Conversions from Signed Integral Types

From	To	Method
char	short	Sign-extend.
char	long	Sign-extend.
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit.
char	unsigned short	Sign-extend to **short**; convert **short** to **unsigned short**.
char	unsigned long	Sign-extend to **long**; convert **long** to **unsigned long**.
char	float	Sign-extend to **long**; convert **long** to **float**.
char	double	Sign-extend to **long**; convert **long** to **double**.
short	char	Preserve low-order byte.
short	long	Sign-extend.
short	unsigned char	Preserve low-order byte.
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit.
short	unsigned long	Sign-extend to **long**; convert **long** to **unsigned long**.
short	float	Sign-extend to **long**; convert **long** to **float**.
short	double	Sign-extend to **long**; convert **long** to **double**.
long	char	Preserve low-order byte.
long	short	Preserve low-order word.
long	unsigned char	Preserve low-order byte.
long	unsigned short	Preserve low-order word.
long	unsigned long	Preserve bit pattern; high-order bit loses function as sign bit.
long	float	Represent as a **float**; if the **long** cannot be represented exactly, some loss of precision occurs.
long	double	Represent as a **double**; if the **long** cannot be represented exactly as a **double**, some loss of precision occurs.

Note: The **int** type is equivalent either to the **short** type or to the **long** type, depending on the implementation. Conversion of an **int** value proceeds as for a **short** or a **long**, whichever is appropriate.

5.7.1.2 Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Unsigned values are converted to floating-point values by first converting to a signed integer of the same size, then converting that signed value to a floating-point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value represented changes if the sign bit is set.

Conversions from unsigned integral types are summarized in Table 5.3.

Table 5.3
Conversions from Unsigned Integral Types

From	To	Method
unsigned char	char	Preserve bit pattern; high-order bit becomes sign bit.
unsigned char	short	Zero-extend.
unsigned char	long	Zero-extend.
unsigned char	unsigned short	Zero-extend.
unsigned char	unsigned long	Zero-extend.
unsigned char	float	Convert to long; convert long to float.
unsigned char	double	Convert to long; convert long to double.
unsigned short	char	Preserve low-order byte.
unsigned short	short	Preserve bit pattern; high-order bit becomes sign bit.
unsigned short	long	Zero-extend.
unsigned short	unsigned char	Preserve low-order byte.
unsigned short	unsigned long	Zero-extend.
unsigned short	float	Convert to long; convert long to float.
unsigned short	double	Convert to long; convert long to double.
unsigned long	char	Preserve low-order byte.
unsigned long	short	Preserve low-order word.
unsigned long	long	Preserve bit pattern; high-order bit becomes sign bit.
unsigned long	unsigned char	Preserve low-order byte.
unsigned long	unsigned short	Preserve low-order word.
unsigned long	float	Convert to long; convert long to float.
unsigned long	double	Convert to long; convert long to double.

Note: The unsigned int type is equivalent either to the unsigned short type or to the unsigned long type, depending on the implementation. Conversion of an unsigned int value proceeds as for an unsigned short or an unsigned long, whichever is appropriate.

5.7.1.3 Conversions from Floating-Point Types

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the value is too large to fit into a **float**, precision is lost.

A floating-point value is converted to an integer value by converting to a **long**. Conversions to other integer types take place as for a **long**. The decimal portion of the floating-point value is discarded in the conversion to a **long**. If the result is still too large to fit into a **long**, the result of the conversion is undefined.

Conversions from floating-point types are summarized in Table 5.4.

Table 5.4
Conversions from Floating-Point Types

From	To	Method
float	char	Convert to **long**; convert **long** to **char**.
float	short	Convert to **long**; convert **long** to **short**.
float	long	Truncate at decimal point; if result is too large to be represented as a **long**, result is undefined.
float	unsigned short	Convert to **long**; convert **long** to **unsigned short**.
float	unsigned long	Convert to **long**; convert **long** to **unsigned long**.
float	double	Change internal representation.
double	char	Convert to **float**; convert **float** to **char**.
double	short	Convert to **float**; convert **float** to **short**.
double	long	Truncate at decimal point; if result is too large to be represented as a **long**, result is undefined.
double	unsigned short	Convert to **long**; convert **long** to **unsigned short**.
double	unsigned long	Convert to **long**; convert **long** to **unsigned long**.
double	float	Represent as a float; if the **double** value cannot be represented exactly as a **float**, loss of precision occurs; if the value is too large to be represented in a **float**, the result is undefined.

5.7.1.4 Conversions from Other Types

An **enum** value is an **int** value, by definition of the **enum** type. Conversions to and from an **enum** value proceed as for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

A pointer value behaves like an unsigned integer value in conversions, with the size of the pointer determined by the implementation. Conversions to and from a pointer type proceed as for an unsigned integer of the appropriate size, except that pointers cannot be converted to floating-point types.

A pointer to one type of value can be converted to a pointer to a different type. The result may be undefined, however, because of the alignment requirements and sizes of different types in storage. In some implementations the **near** and **far** keywords modify pointer sizes. Conversions between **near** and **far** pointers may produce meaningless addresses.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to **void** by assignment. However, a value can be explicitly cast to **void**, as discussed in Section 5.7.2, "Type-Cast Conversions."

5.7.2 Type-Cast Conversions

Explicit type conversions can be made by means of a type cast. A type cast has the form

(type-name)operand

where *type-name* specifies a particular type and *operand* is a value to be converted to the specified type. (Type names are discussed in Section 4.9 of Chapter 4, "Declarations.")

The conversion of *operand* takes place as though it had been assigned to a variable of the named type. The conversion rules for assignments (outlined in Section 5.7.1) apply to type casts as well. The type name **void** can be used in a cast operation, but the resulting expression cannot be assigned to any item.

5.7.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand and operands. Many operators perform the "usual arithmetic conversions," which are outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions. See the discussions of additive operators (Section 5.3.6) and subscript expressions (Section 5.2.5) for details.

5.7.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on whether a forward declaration with declared argument types is present for the called function.

If a forward declaration is present, and it includes declared argument types, the compiler performs type-checking. The type-checking process is outlined in detail in Section 7.4.1, "Actual Arguments," of Chapter 7, "Functions."

If no forward declaration is present, or if the forward declaration omits the argument type list, the only conversions performed on the arguments in the function call are the usual arithmetic conversions. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

Chapter 6
Statements

6.1 Introduction 133
6.2 Break Statement 135
6.3 Compound Statement 136
6.4 Continue Statement 138
6.5 Do Statement 139
6.6 Expression Statement 140
6.7 For Statement 141
6.8 Goto and Labeled Statements 143
6.9 If Statement 145
6.10 Null Statement 147
6.11 Return Statement 148
6.12 Switch Statement 150
6.13 While Statement 153

6.1 Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows.

break statement

compound statement

continue statement

do statement

expression statement

for statement

goto statement

if statement

null statement

return statement

switch statement

while statement

C statements consist of keywords, expressions, and other statements. The keywords that appear in C statements are

break	do	if
case	else	return
continue	for	switch
default	goto	while

The expressions in C statements are the expressions discussed in Chapter 5, "Expressions and Assignments." Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the "body" of the enclosing statement. Frequently the statement body is a "compound" statement, that is, a single statement composed of one or more statements.

The compound statement is delimited by braces. All other C statements end with a semicolon.

Any C statement may be prefixed with an identifying label consisting of a name and a colon. Statement labels are recognized only by the **goto** statement and are therefore discussed with the **goto** statement.

When a C program is executed, its effect is that of the execution of the statements in order of their appearance in the program, except where a statement explicitly transfers control to another location.

6.2 Break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **break**, **switch**, or **while** statement in which it appears. Control passes to the statement following the terminated statement. A **break** statement appearing outside any **do**, **for**, **switch**, or **while** statement causes an error.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement immediately enclosing it. To transfer control out of the nested structure altogether, a **return** or **goto** statement can be used.

Example

```
for (i = 0; i < LENGTH - 1; i++) {
        for (j = 0; j < WIDTH - 1; j++) {
                if (lines[i][j] == '\0') {
                        lengths[i] = j;
                        break;
                }
        }
}
```

The above example processes an array of variable length strings stored in *lines*. The **break** statement causes an exit from the interior **for** loop after the terminating null character (`'\0'`) of each string is found and stored in *lengths*`[`*i*`]`. Control then returns to the outer **for** loop. The variable *i* is increased and the process is repeated until *i* is greater than or equal to LENGTH−1.

6.3 Compound Statement

Syntax

{
[*declaration*]
.
.
.
statement
[*statement*]
.
.
.
}

Execution

The effect of a compound statement's execution is that of the execution of its statements in order of their appearance, except where a statement explicitly transfers control to another location.

Example

```
if (i > 0) {
        line[i] = x;
        x++;
        i--;
}
```

A compound statement typically appears as the body of another statement such as the **if** statement. In the above example, if *i* is greater than zero, all of the statements in the compound statement are executed in order.

Labeled Statements

Like other C statements, any of the statements in a compound statement may carry a label. Transfer into the compound statement by means of a **goto** is therefore possible. However, transferring into a compound statement is dangerous when the compound statement includes declarations

that initialize variables. Declarations in a compound statement precede the executable statements, so transferring directly to an executable statement within the compound statement bypasses the initializations. The results are unpredictable.

6.4 Continue Statement

Syntax

continue;

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. Within a **do** or a **while** statement, the next iteration begins with the reevaluation of the **do** or **while** statement's expression. Within a **for** statement, the next iteration starts with the evaluation of the **for** statement's *loop-expression*. It proceeds with the evaluation of the conditional expression and subsequent termination or reiteration of the statement body.

Example

```
while (i-- > 0) {
        x = f(i);
        if (x == 1)
                continue;
        y = x * x;
}
```

The statement body is executed if *i* is greater than zero. First "f(i)" is assigned to *x*. Then, if *x* is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of "i-- > 0".

6.5 Do Statement

Syntax

do
statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false. First, the statement body is executed. Then *expression* is evaluated. If *expression* is false (zero), the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the statement body is executed again, and *expression* is tested again. The statement body is executed repeatedly until *expression* becomes false.

The **do** statement may also terminate with the execution of a **break**, **goto**, or **return** statement within the statement body.

Example

```
do {
        y = f(x);
        x--;
} while (x > 0);
```

The two statements "y = f(x);" and "x--;" are executed, regardless of the initial value of *x*. Then "x > 0" is evaluated. If *x* is greater than zero, the statement body is executed again and "x > 0" is reevaluated. The statement body is executed repeatedly so long as *x* remains greater than zero. Execution of the **do** statement terminates when *x* becomes zero or negative.

6.6 Expression Statement

Syntax

expression;

Execution

The expression is evaluated, according to the rules outlined in Chapter 5, "Expressions and Assignments."

Examples

```
1.  x = (y + 3);

2.  x++;

3.  f(x);
```

In C, assignments are expressions; the value of the expression is the value being assigned (sometimes called the "right-hand value"). In the first example, x is assigned the value of "y + 3". In the second example, x is increased by 1.

The third example shows a function call expression. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually incorporates an assignment to store the returned value when the function is called. If the return value is not assigned, as in the example, the function call is executed but the return value, if any, is not used.

6.7 For Statement

Syntax

for ([*init-expression*]; [*cond-expression*]; [*loop-expression*])
statement;

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. The *init-expression* and *loop-expression* are optional expressions that can be used to initialize and modify values during the **for** statement's execution.

The first step in the execution of the **for** statement is the evaluation of *init-expression*, if present. Next, *cond-expression* is evaluated, with three possible results.

1. If the conditional expression is true (nonzero), the statement body is executed; then *loop-expression*, if present, is evaluated; then the process begins again with the evaluation of *cond-expression*.
2. If the conditional expression is omitted, the conditional expression is considered true; execution proceeds exactly as described above. A **for** statement lacking *cond-expression* terminates only upon the execution of a **break**, **goto**, or **return** statement within the statement body.
3. If the conditional expression is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement may also terminate with the execution of a **break**, **return**, or **goto** statement within the statement body.

Example

```
for (i = space = tab = 0; i < MAX; i++) {
        if (line[i] == '\0x20')
                space++;
        if (line[i] == '\t') {
                tab++;
                line[i] ='\0x20';
        }
}
```

The above example counts space (`'\0x20'`) and tab (`'\t'`) characters in the array of characters named *line* and replaces each tab character with a space. First *i*, *space*, and *tab* are initialized to zero. Then *i* is compared to the constant MAX; if *i* is less than MAX, the statement body is executed. Depending on the value of *line*`[i]`, the body of one or neither of the if statements is executed. Then *i* is increased and tested against MAX. The statement body is executed repeatedly as long as *i* is less than MAX.

6.8 Goto and Labeled Statements

Syntax

goto *name*;

.
.
.

name: *statement*

Execution

The **goto** statement transfers control directly to the statement specified by *name*. The labeled statement is executed immediately after the **goto** statement is executed. An error results if no statement with the given label resides in the same function or if an identical label appears before more than one statement in the same function.

A statement label is meaningful only to a **goto** statement. When a labeled statement is encountered in any other context, the statement is executed without regard to the label.

Example

```
if (errorcode > 0)
        goto exit;
            .
            .
            .
exit:
        return (errorcode);
```

In the example, a **goto** statement transfers control to the point labeled *exit* when an error occurs.

Forming Labels

A label name is simply an identifier, formed by following the same rules that govern the construction of identifiers (see Section 2.4 of Chapter 2, "Elements of C"). Each statement label must be distinct from other statement labels and identifiers in the same function.

6.9 If Statement

Syntax

```
if (expression)
        statement1
[else
        statement2 ]
```

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*. First, *expression* is evaluated. If *expression* is true (nonzero), the statement immediately following it is executed. If *expression* is false, the statement following the **else** keyword is executed. If *expression* is false and the **else** clause is omitted, the statement following *expression* is ignored. Control then passes from the **if** statement to the next statement in the program.

Example

```
if (i > 0)
        y = x/i;
else {
        x = i;
        y = f(x);
}
```

In the example, the statement "y = x/i;" is executed if *i* is greater than zero. If *i* is less than or equal to zero, *i* is assigned to *x* and "f(x)" is assigned to *y*. Notice that the statement forming the **if** clause ends with a semicolon.

Nesting

C does not offer an "else if" statement, but the same effect is achieved by nesting **if** statements. An **if** statement may be nested in either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. In the absence of braces, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Examples

```
1.  if (i > 0)                  /* Without braces */
        if (j > i)
                x = j;
        else
                x = i;
```

```
2.  if (i > 0) {                /* With braces */
        if (j > i)
                x = j;
}
else
        x = i;
```

In the first example, the **else** is associated with the inner **if** statement. If *i* is less than or equal to zero, no value is assigned to *x*.

In the second version, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If *i* is less than or equal to 0, *i* is assigned to *x*.

6.10 Null Statement

Syntax

;

Execution

A null statement is a statement containing only a semicolon. It may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Example

```
for (i = 0; i < 10; line[i++] = 0)
        ;
```

Statements such as **do**, **for**, **if**, and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body. In the above example, the third expression of the **for** statement initializes the first ten elements of *line* to zero. The statement body is a null statement, since no further statements are necessary.

Labeling a Null Statement

The null statement, like any other C statement, may be prefixed by an identifying label. To label an item that is not a statement, such as the closing brace of a compound statement, you can insert and label a null statement immediately before the item to get the same effect.

6.11 Return Statement

Syntax

return [*expression*];

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point just after the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

Example

```
main()
{
                .
                .
                .
        y = sq(x);
        draw(x, y);
                .
                .
                .
}

sq(x)
int x;
{
        return (x * x);
}

void draw(x,y)
int x, y;
{
                .
                .
                .
        return;
}
```

The *main* function calls two functions, *sq* and *draw*. The *sq* function returns the value of "x * x" to *main*. The return value is assigned to *y*. The *draw* function is declared as a **void** function and does not return a value. An attempt to assign the return value of *draw* would cause an error.

By convention, parentheses enclose the *expression* of the **return** statement, as shown above. The language does not require the parentheses.

Omitting the Return Statement

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function. The return value of the called function is undefined. If a return value is not required, the function should be declared to have **void** return type.

6.12 Switch Statement

Syntax

```
switch (expression) {
        [declaration]
                .
                .
                .
        [case constant-expression :]
                .
                .
                .
                [statement]
                .
                .
                .
        [default :
                statement]
        [case constant-expression :]
                .
                .
                .
                [statement]
                .
                .
                .
}
```

Execution

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose **case** *constant-expression* matches the value of the *expression* in parentheses. Execution of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of the **switch** *expression*. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body are executed.

The **switch** expression must be an integral or **enum** value. If the *expression* is shorter than an **int**, it is widened to an **int** value. Each **case** *constant-expression* is then cast to the type of the **switch** expression. The value of each **case** *constant-expression* must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines the starting point for execution of the statement body. All statements appearing between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The effect of the **switch** statement is to transfer control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples

```
1.  switch (c) {
            case 'A':
                    capa++;
            case 'a':
                    lettera++;
            default :
                    total++;
          }

2.  switch (i) {
            case -1:
                    n++;
                    break;
            case 0 :
                    z++;
                    break;
            case 1 :
                    p++;
                    break;
          }
```

In the first example, all three statements of the **switch** body are executed if *c* is equal to 'A'. Execution control is transferred to the first statement ("capa++;") and continues in order through the rest of the body. If *c* is equal to 'a', *lettera* and *total* are increased. Only *total* is increased if *c* is not equal to 'A' or 'a'.

In the second example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the **switch** after one statement in the body is executed. If *i* is equal to −1, only *n* is increased. The **break** following the statement "n++;" causes execution control to pass out of the **switch** body, bypassing the remaining statements. Similarly, if *i* is equal to 0, only *z* is increased; if *i* is equal to 1, only *p* is increased. The final **break** statement is not strictly necessary, since control will pass out of the body at the end of the compound statement, but it is included for consistency.

Multiple Labels

A statement may carry multiple **case** labels, as the following sample shows.

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although any statement within the body of the **switch** statement may be labeled, no statement is required to carry a label. Statements without labels may be freely intermingled with labeled statements. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all succeeding statements in the block are executed, regardless of their labels.

6.13 While Statement

Syntax

while (*expression*)
statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false. First, *expression* is evaluated. If the *expression* is initially false (zero), the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program. If *expression* is true (nonzero), the body of the statement is executed. Following each execution of the statement body, *expression* is reevaluated. The body is executed repeatedly as long as *expression* remains true.

The **while** statement may also terminate with the execution of a **break**, **goto**, or **return** within the statement body.

Example

```
while (i >= 0) {
        string1[i] = string2[i];
        i--;
}
```

The above example copies characters from *string2* to *string1*. If *i* is greater than or equal to zero, *string2*[*i*] is assigned to *string1*[*i*] and *i* is decreased. When *i* reaches or falls below 0, execution of the **while** statement terminates.

Chapter 7
Functions

7.1 Introduction 157
7.2 Function Definitions 157
7.2.1 Storage Class 158
7.2.2 Return Type 158
7.2.3 Formal Parameters 161
7.2.4 Function Body 164
7.3 Function Declarations 165
7.4 Function Calls 167
7.4.1 Actual Arguments 170
7.4.2 Calls with a Variable Number of Arguments 172
7.4.3 Recursive Calls 174

7.1 Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one main function and may have other functions. The sections of this chapter describe how to define, declare, and call C functions.

A function *definition* specifies the name of the function, its formal parameters, and the declarations and statements that define its action. The function definition can also give the return type of the function and its storage class.

A function *declaration* establishes the name, return type, and storage class of a function whose explicit definition is given at another point in the program. The number and types of arguments to the function can also be specified in the function declaration. This allows the compiler to compare the types of the actual arguments and the formal parameters of a function. Function declarations are optional for functions whose return type is **int**. To ensure correct behavior, functions with other return types must be declared before they are called.

A function *call* passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

7.2 Function Definitions

A function definition specifies the name, formal parameters, and body of a function. It may also define the function's return type and storage class. A function definition has the following form.

[*sc-specifier*] [*type-specifier*] *declarator* ([*parameter-list*])
[*parameter-declarations*]
function-body

The *sc-specifier* gives the function's storage class, which must be either **static** or **extern**. The *type-specifier* and *declarator* together specify the function's return type and name. The *parameter-list* is a list (possibly

empty) of formal parameters to be used by the function. The *parameter-declarations* establish the types of the formal parameters. The *function-body* is a compound statement containing local variable declarations and statements. The following sections describe the parts of the function definition in detail.

7.2.1 Storage Class

The storage class specifier in a function definition gives the function either **static** or **extern** storage class. A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that constitute the program.

The storage class specifier is required in a function definition in only one case: when the function is declared elsewhere in the same source file with the **static** storage class specifier.

The **static** storage class specifier can also be used when defining a function previously declared in the same source file without a storage class specifier. Normally, a function declared without a storage class specifier defaults to the **extern** class. However, if the function definition explicitly specifies the **static** class, the function is given **static** class instead.

When the storage class specifier is omitted from a function definition, the storage class defaults to **extern**. The **extern** storage class specifier can be explicitly specified in the function definition, but it is not required.

7.2.2 Return Type

The return type of a function defines the size and type of value returned by the function. The type declaration has the form

[*type-specifier*] *declarator*

where *type-specifier*, together with the *declarator*, define the function's return type and name. If no *type-specifier* is given, the return type **int** is assumed.

The *type-specifier* can specify any fundamental, structure, or union type. The *declarator* consists of the function identifier, possibly modified to declare a pointer type. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. Functions with **int** return type do not have to be declared before they are called. Functions with other return types cannot be called before they are either defined or declared.

A function's return value type is used only when the function returns a value. A function returns a value when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point of call. If no **return** statement is executed, or if the executed **return** statement does not contain an expression, the return value of the function is undefined. If the calling function expects a return value, the behavior of your program is also undefined.

Examples

```
1.    /* return type is int */
    static add (x, y)
    int x, y;
    {
        return (x+y);
    }

2.  typedef struct  {
        char name[20];
        int id;
        long class;
    } STUDENT;

        /* return type is STUDENT */
    STUDENT sortstu (a, b)
    STUDENT a, b;
{
    return ( (a.id < b.id) ? a : b );
}

3.    /* return type is char pointer */
    char *smallstr(s1, s2)
    char s1[], s2[];
    {
        int i;

        i=0;
        while ( s1[i] != '\0' && s2[i] != '\0' )
                        i++;
        if ( s1[i] == '\0' )
                return (s1);
        else
                return (s2);
    }
```

In the first example, the return type of *add* is **int** by default. The function has **static** storage class, which means it can be called only by functions in the same source file.

The second example defines the STUDENT type with a **typedef** declaration and defines the function *sortstu* to have STUDENT return type. The function selects and returns one of its two structure arguments.

The third example defines a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements. Thus, the return type of the function is pointer to **char**.

7.2.3 Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. The formal parameters are declared in a parameter list at the beginning of the function declaration. The parameter list defines the names of the parameters and the order in which they take on values in the function call.

The parameter list has the form

([*identifier*[, *identifier*]] ...)

where each *identifier* names a parameter. The parentheses are required.

Parameter declarations define the type and size of values stored in the formal parameters. These declarations have the same form as other variable declarations (see Section 4.4 of Chapter 4, "Declarations"). A formal parameter can have any fundamental, structure, union, pointer, or array type.

A parameter can only have **auto** or **register** storage class. If no storage class is given, **auto** storage is assumed. If a formal parameter is named in the parameter list but is not declared, the parameter is assumed to have **int** type. Formal parameters can be declared in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be used for variable declarations within the function body.

The type of the formal parameter should correspond to the type of the actual argument and to the type of the corresponding argument in the argument type list for the function, if present. If the function has a variable number of arguments, the user is responsible for determining the number of arguments passed and for retrieving additional arguments from the stack within the body of the function.

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it an **int**.

The converted type of each formal parameter determines how the arguments placed on the stack by the function call are interpreted. A type mismatch between an actual and a formal parameter can cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are stored in the **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. Errors of this kind can be detected through diligent use of argument type lists in function declarations.

Example

```
struct student {
        char name[20];
        int id;
        long class;
        struct student *nextstu;
} student;

main()
{
        int match ( struct student *, char * );
                .
                .
                .
        if (match (student.nextstu, student.name) > 0) {
                .
                .
                .
        }
}

match ( r, n )
struct student *r;
char *n;
{
        int i = 0;

        while ( r->name[i] == n[i] )
                if ( r->name[i++] == '\0' )
                        return (r->id);
        return (0);
}
```

The example contains a structure type declaration, a forward declaration of the function *match*, a call to *match,* and the definition of the *match* function. Notice that the same name, *student,* can be used without conflict both for the structure tag and for the structure variable name.

The *match* function is declared to have two arguments, the first a pointer to the *student* structure type and the second a pointer to a **char** type.

The two formal parameters of the *match* function are *r* and *n*. The parameter *r* is declared as a pointer to the *student* structure type. The parameter *n* is declared as a pointer to a **char** type.

The function is called with two arguments, both members of the *student* structure. Because there is a forward declaration of *match*, the compiler performs type-checking between the actual arguments and the argument type list and between the actual arguments and the formal parameters. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer, and is used in subscripted expressions as though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as "char *n" is the same as declaring it "char n[]".

Within the function, the local variable *i* is defined and used to keep track of the current position in the array. The function returns the *id* structure member if the *name* member matches the array *n*. Otherwise, it returns zero.

7.2.4 Function Body

The function body is simply a compound statement. The compound statement contains the statements that define the function's action and can also contain declarations of variables used by these statements. See Section 6.3 of Chapter 6, "Statements," for a discussion of compound statements.

All variables declared in the function body have **auto** storage type unless otherwise specified. When the function is called, storage space for the local variables is created and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement or the end of the function body is encountered. Control then passes back to the point of call.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include the optional expression.

7.3 Function Declarations

A function declaration defines the name, return type, and storage class of a given function, and may establish the type of some or all of the function's arguments. See Chapter 4, "Declarations," for a detailed description of the syntax of function declarations.

Functions can be declared implicitly or with forward declarations. The return type of a function declared either implicitly or with a forward declaration must agree with the return type specified in the function definition.

An implicit declaration occurs whenever a function is called without being previously defined or declared. The C compiler implicitly declares the function to have **int** return type. By default, the function is declared to have **extern** storage class. The function definition can redefine the storage class to **static**, provided the function definition is given later in the same source file.

A forward declaration establishes the attributes of a function, allowing the declared function to be called before it is defined or to be called from another source file. If the storage class specifier **static** is given in a forward declaration, the function has **static** class. The function definition must also specify the **static** class. If the storage class specifier is **extern** or is omitted, the function has **extern** class. However, the function definition can redefine the storage class as **static**, provided the function definition appears below the declaration in the same source file.

Forward declarations have several important uses. They establish the return type for functions that return any type of value but **int**. (Functions that return **int** values can also have forward declarations, but do not require them.) Functions with non-**int** return types cannot be called before they are either declared or defined; the compiler assumes that the called function has **int** return type.

Forward declarations can be used to establish the types of arguments expected in a function call. The optional argument type list of a forward declaration gives the type and number of arguments expected. (The number of arguments can be variable.) The argument type list is a list of type names corresponding to the expression list in the function call.

If no argument type list is supplied, no type-checking is performed. Type mismatches between actual arguments and formal parameters are silently accepted. Type-checking is discussed further in Section 7.4.1, "Actual Arguments."

Forward declarations are also used to declare pointers to functions before the functions are defined.

Example

```
main()
{
        int a = 0, b = 1;
        float x = 2.0, y = 3.0;
        double realadd(double, double);

        a = intadd (a, b);
        x = realadd(x, y);
}

intadd(a, b)
int a, b;
{
        return (a + b);
}

double realadd(x, y)
double x, y;
{
        return (x + y);
}
```

In the example, the function *intadd* is implicitly declared to return an **int** value, since it is called before it is defined. The compiler does not check the types of the arguments in the call because no argument type list is available.

The function *realadd* returns a **double** value instead of an **int**. The forward declaration of *realadd* in the *main* function allows the *realadd* function to be called before it is defined. Notice that the definition of *realadd* matches the forward declaration by specifying the **double** return type.

The forward declaration of *realadd* also establishes the type of its two arguments. The actual arguments match the types given in the forward declaration and also match the types of the formal parameters.

7.4 Function Calls

A function call is an expression that passes control and zero or more actual arguments to a function. A function call has the form

expression (*expression-list*)

where *expression* evaluates to a function address and *expression-list* is a list of expressions whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

When the function call is executed, the expressions in the function expression list are copied, converted as necessary, and then passed to formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and so on through the end of the list. Since the called function works with copies of the actual arguments, any changes it makes to the arguments are not reflected in the original values from which the copies were made.

Execution control then passes to the first statement in the function. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed. The return value is undefined.

The expressions in the function call's expression list can be evaluated in any order, so expressions with side effects have unpredictable results. The only guarantee the compiler makes is that all side effects in the expression list are evaluated before control passes to the called function.

The only requirement in calling a function is for the expression before the parentheses to evaluate to a function address. This means that a function can be called through any function pointer expression. It may be helpful to remember that a function is called in the same manner it is declared. For instance, when declaring a function, the name of the function is given, followed by an argument type list in parentheses. To call the function, only the name of the function is required, followed by an expression list in parentheses. The indirection operator (*) is not required to call the function; the name of the function evaluates to the function address, which is used to call the function.

The same principle applies when calling a function through a pointer. For example, suppose a function pointer is declared as follows.

```
int (*fpointer)(int, int);
```

The identifier *fpointer* is declared to point to a function taking two **int** arguments and returning an **int** value. A function call through *fpointer* might look like this:

```
(*fpointer)(3,4)
```

The indirection operator (*) is used to obtain the address of the function to which *fpointer* points. The function address is then used to call the function.

Examples

```
1.  double *realcomp(double, double);
    double a, b, *rp;
                    .
                    .
                    .
    rp = realcomp(a, b);

2.  main ()
    {
        long lift(int), step(int), drop(int);
        void work (int, long (*)(int));
        int select, count;
                    .
                    .
                    .
        select = 1;
        switch ( select ) {
                case 1: work(count, lift);
                        break;

                case 2: work(count, step);
                        break;

                case 3: work(count, drop);

                default:
                        break;
        }
}

void work ( n, func )
int n;
long (*func)(int);
{
        int i;
        long j;

        for (i = j = 0; i < n; i++)
                j += (*func)(i);
}
```

In the first example, the *realcomp* function is called in the statement "rp = realcomp(a, b);". Two **double** arguments are passed to the *realcomp* function. The return value, a pointer to a **double**, is assigned to *rp*.

In the second example, the function call

```
work (count, lift);
```

in *main* passes an integer variable and the address of the function *lift* to the function *work*. Notice that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used. Otherwise, the identifier is not recognized. In this case, a forward declaration for *work* is given at the beginning of the *main* function.

The formal parameter *func* in *work* is declared to be a pointer to a function taking one **int** argument and returning a **long**. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long**.

The function *work* calls the selected function by using the function call

```
(*func)(i);
```

One argument, *i*, is passed to the called function.

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although arrays and functions cannot be passed as parameters, pointers to these items can be passed.

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variables from which it was originally derived.

Pointers provide a way to access a value by reference from a function. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

Each expression in a function call is evaluated and converted as follows. If an argument type list for the called function is available, the usual arithmetic conversions are performed independently on each expression in the expression list and on each type in the argument type list. Each expression in the expression list is then compared with the type name that

occupies the corresponding position in the argument type list. The value of the expression is converted (if necessary) to the named type as if by assignment.

Next, the converted expression is compared with the type of the formal parameter that has the same place in the parameter list as the expression has in the expression list. (The formal parameters also undergo the usual arithmetic conversions before the comparison.) No conversions are performed, but the compiler produces warning messages as if the expressions were assigned to the formal parameters.

The number of expressions given in the expression list must match the number of formal parameters, unless the function's forward declaration explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the argument type list and converts them, if necessary, as described above. If there are additional actual arguments in the function call, each additional argument undergoes the usual arithmetic conversions, but is not otherwise converted or checked.

If the argument type list contains the special type name **void**, the compiler expects zero actual arguments in the function call and zero formal parameters. It produces a warning message if it finds otherwise.

If the argument type list is empty (omitted) or the called function has no forward declaration, the compiler performs no type-checking, either for type or for number of arguments. In this case, the actual arguments in the function call, if any, undergo the usual arithmetic conversions independently before they are placed on the stack.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted. If the type of the formal parameter does not match the type of the actual argument, the data on the stack can be misinterpreted.

Type mismatches between actual and formal parameters can produce serious errors, particularly when the mismatches entail size differences. Keep in mind that these errors are not detected unless an argument type list is given in the forward declaration of the function.

Example

```
main ()
{
        void swap (int *, int *);
        int x, y;
           .
           .
           .
        swap(&x, &y);
}

void swap (a, b)
int *a, *b;
{
        int t;

        t = *a;
        *a = *b;
        *b = t;
}
```

In the above example, the *swap* function is declared in *main* to have two arguments, both pointers to integers. The formal parameters *a* and *b* are also declared as pointers to integer variables. In the function call

```
swap (&x, &y)
```

the address of *x* is stored in *a* and the address of *y* is stored in *b*. There are two names, or "aliases," for the same location. References to **a* and **b* in *swap* are effectively references to *x* and *y* in *main*. The assignments within *swap* change the contents of *x* and *y*.

The compiler performs type-checking on the arguments to *swap* because an argument type list is present in the forward declaration of *swap*. The types of the actual arguments match both the argument type list and the formal parameters.

7.4.2 Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, the programmer simply gives any number of arguments in the function call. In the forward declaration of the function (if there is one), a variable number of arguments is specified by placing a comma at the end of the argument type list (see Section 4.5 of Chapter 4, "Declarations"). One argument must be present in the function call for each type name specified in the argument

type list. If only a comma (but no type names) are given, no arguments are required when calling the function.

All the arguments given in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. The programmer is responsible for retrieving any additional arguments from the stack and for determining how many arguments are present.

Example

```
main()
{
        int scores (int, );
        int count, average, i;
           .
           .
           .
        average = scores (count, 14, 96, 82);
           .
           .
           .
}

scores (number)
int number;
{
        int *ip, total = 0, i;
        ip = &number + 1;

        for (i = 1; i <= number; i++, ip++)
                total += *ip;

        if (number > 0)
                return (total/number);
        return (-1);
}
```

The above example shows a function named *scores* that takes a variable number of arguments. The forward declaration of *scores* in *main* establishes that *scores* has at least one argument, an **int**. The comma at the end of the argument type list means that there may be more undeclared arguments.

In the call to *scores*, four actual arguments are passed. The first argument is checked for compatibility with the argument type list and the formal parameter of *scores*. Since the types match, no conversions or warning messages are necessary.

In the definition of the *scores* function, one formal parameter is declared. The additional arguments are retrieved by taking the address of the previous argument (*number* in the first case), increasing it, and retrieving the value at that address. This procedure works because arguments passed to a function are stored in order on the stack.

The *number* argument is assumed to hold the number of additional arguments, so its value controls how many additional arguments are retrieved from the stack. When *number* arguments have been retrieved and added to the total, the average of the scores is returned to the *main* function. The value −1 is returned if *number* is zero.

7.4.3 Recursive Calls

Any function in a C program can be called recursively. A function can therefore call itself. The C compiler allows any number of recursive calls to a function. On each call, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Previous parameters are inaccessible to all versions of the function except the version in which they were created.

Notice that variables declared with global storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler defines no limit on the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.

Chapter 8
Preprocessor Directives

8.1 Introduction 177
8.2 Manifest Constants and Macros 177
8.2.1 Define Directive 178
8.2.2 Undefine Directive 181
8.3 Include Files 182
8.4 Conditional Compilation 184
8.4.1 If, Elif, Else, and Endif Directives 184
8.4.2 Ifdef and Ifndef Directives 188
8.5 Line Control 188

8.1 Introduction

The C preprocessor is a text processor used to manipulate the text of a source file before compilation. The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling. This chapter explains the main tasks performed by preprocessor directives and describes each directive in detail.

Preprocessor directives are typically used to make source programs easy to modify and to compile in different execution environments. Directives in the source file instruct the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, and suppress compilation of a portion of the file by removing blocks of text.

The C preprocessor recognizes the following directives.

#define	**#ifdef**
#elif	**#ifndef**
#else	**#include**
#endif	**#line**
#if	**#undef**

The number sign (**#**) must be the first nonwhitespace character on the line containing the directive. Whitespace characters can appear between the number sign and the first letter of the directive. Some directives are followed by arguments or values, as described below. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

8.2 Manifest Constants and Macros

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "manifest constants." Identifiers that represent statements or expressions are called "macros."

Once an identifier is defined, it cannot be redefined to a different value without first removing the definition. However, the identifier can be redefined with exactly the same definition. Thus, a program is allowed to contain more than one occurrence of the same definition.

The **#undef** directive removes the definition of an identifier. Once the definition has been removed, the identifier can be redefined to a different value. Sections 8.2.1 and 8.2.2 discuss the **#define** and **#undef** directives respectively.

Macros can be defined to look and act like function calls. Because macros do not generate actual function calls, replacing function calls with macros can improve execution time. However, macros create problems if they are not defined and used with care. Macro definitions with arguments may require the use of parentheses to preserve the proper precedence in an expression. In addition, macros may not handle expressions with side effects correctly. See the examples in Section 8.2.1 for details.

8.2.1 Define Directive

Syntax

#define *identifier text*
#define *identifier*(*parameter-list*) *text*

The **#define** directive substitutes the given *text* for subsequent occurrences of the specified *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in Chapter 2 and in Appendix B.) For instance, the *identifier* is not replaced when it occurs within strings or as part of a longer identifier.

If a *parameter-list* appears after the *identifier*, the **#define** directive replaces each occurrence of *identifier*(*argument-list*) with a version of *text* modified by substituting actual arguments for formal parameters.

The *text* consists of a series of tokens, such as keywords, constants, or complete statements. One or more whitespace characters must separate the *text* from the *identifier* (or from the closing parenthesis of the *parameter-list*). If the text is longer than one line, it can be continued onto the next line by preceding the newline character with a backslash (\).

The *text* can also be empty. The effect of this option is to remove instances of the given *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if** directive (discussed later in this chapter).

The *parameter-list*, when given, consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces between the *identifier* and the opening parenthesis are allowed.

Formal parameter names appear in *text* to mark the places where actual values will be substituted. Each parameter name can occur more than once in the *text*, and the names can appear in any order.

The actual arguments following an instance of the *identifier* in the source file are matched to the formal parameters of the *parameter-list*, and the *text* is modified by replacing each formal parameter with the corresponding actual argument. The actual *argument-list* and the formal *parameter-list* must have the same number of arguments.

Arguments with side effects sometimes cause macros to produce unexpected results. A macro definition may contain more than one occurrence of a given formal parameter. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, is evaluated more than once (see Example 4 below).

Examples

```
1.  #define WIDTH        80
    #define LENGTH       (WIDTH + 10)

2.  #define FILEMESSAGE "Attempt to create file \
    failed because of insufficient space"

3.  #define REG1         register
    #define REG2         register
    #define REG3

4.  #define MAX(x,y)     ((x) > (y)) ? (x) : (y)

5.  #define MULT(a,b)    ((a) * (b))
```

The first example defines the identifier WIDTH as the integer constant 80, and defines LENGTH in terms of WIDTH and the integer constant 10. Each occurrence of LENGTH is replaced with "(WIDTH + 10)", which is

in turn replaced with the expression "(80 + 10)". The parentheses around "WIDTH + 10" are important because they control the interpretation in a statement such as the following.

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes

```
var = (80 + 10) * 20;
```

or 1800. Without parentheses, the result is

```
var = 80 + 10 * 20;
```

which evaluates to 280 because the multiplication operator (*) has higher precedence than the addition operator (+).

The second example defines the identifier FILEMESSAGE. The definition is extended to a second line by using the backslash escape character (\).

The third example defines three identifiers, REG1, REG2, and REG3. REG1 and REG2 are defined as the keyword **register**. The definition of REG3 is empty, so each occurrence of REG3 is removed from the source file. These directives can be used to ensure that the program's most important variables (declared with REG1 and REG2) are given **register** storage. See the discussion of the **#if** directive later in Section 8.4.1 for an expanded version of this example.

The fourth example defines a macro named MAX. Each occurrence of the identifier MAX following the definition in the source file is replaced by the expression "((x) > (y)) ? (x) : (y)", where actual values replace the parameters *x* and *y*. For example, the occurrence

```
MAX(1,2)
```

is replaced with

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i,s[i])
```

is replaced with

```
((i) > (s[i])) ? (i) : (s[i])
```

This macro is easier to read than the corresponding expression, making the source program easier to understand.

Notice that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence "MAX(i, s[i++])" is replaced with "((i) > (s[i++])) ? (i) : (s[i++])". The expression "(s[i++])" is evaluated twice, so by the time the ternary expression has been fully evaluated, *i* has increased by two. The result of the ternary expression is unpredictable, since the operands of the ternary expression can be evaluated in any order, and the value of *i* varies depending on the evaluation order.

The fifth example defines the macro MULT. Once the macro is defined, an occurrence such as "MULT(3, 5)" is replaced by "(3) * (5)". The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro. For instance, the occurrence "MULT(3 + 4, 5 + 6)" is replaced by "(3 + 4) * (5 + 6)", which evaluates to 77. Without the parentheses, the result is "3 + 4 * 5 + 6", which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

8.2.2 Undefine Directive

Syntax

#undef *identifier*

The **#undef** directive removes the current definition of *identifier*. The preprocessor ignores subsequent occurrences of *identifier*. To remove a macro definition using **#undef**, give only the macro *identifier*. Do not give a parameter list.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see Section 8.4.1) to control compilation of portions of the source program.

Example

```
#define WIDTH           80
#define ADD(X,Y)        (X) + (Y)
   .
   .
   .
#undef WIDTH
#undef ADD
```

In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given. The **#undef** directive can also be applied to an identifier that has no previous definition. This ensures that the identifier is undefined.

8.3 Include Files

Syntax

#include "*pathname*"
#include <*pathname*>

The **#include** directive adds the contents of a given "include file" to another file. Constant and macro definitions can be organized into include files and added to any source file by using **#include** directives. Include files are also useful for incorporating declarations of external variables and complex data types. The types need only be defined and named once in an include file created for that purpose.

The **#include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point of the directive. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *pathname* is a filename optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the specific operating system on which the program is compiled.

The preprocessor uses the concept of a "standard" directory or directories to search for included files. The location of the standard directories for include files depends on the implementation and the operating system. See your system documentation for a definition of the standard directories.

The preprocessor stops searching as soon as it finds a file with the given name. If a complete, unambiguous pathname for the include file is given, either in double quotation marks (" ") or in angle brackets (< >), the preprocessor searches only that pathname and ignores the standard directories.

If the file specification does not give a complete pathname, and the file specification is enclosed in double quotation marks, the preprocessor searches for the file in the same directory as the including file first (the "current working directory"). It then searches directories specified in the compiler command line and finally searches the standard directories.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in directories specified in the compiler command line and then searches the standard directories.

An **#include** directive can be nested. In other words, the directive can appear in a file named by another **#include** directive. When the preprocessor encounters the nested **#include** directive, it processes the named file and inserts it into the current file. The preprocessor uses the same search procedures outlined above in searching for nested include files.

The new file can also contain **#include** directives. Nesting can continue up to ten levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Examples

```
1.  #include <stdio.h>

2.  #include "defs.h"
```

The first example adds the contents of the file named *stdio.h* to the source program. The angle brackets cause the preprocessor to search the standard directories for *stdio.h*, after searching directories specified in the command line.

The second example adds the contents of the file specified by *defs.h* to the source program. The double quotation marks mean that the directory containing the current source file is searched first.

8.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow for suppressing compilation of portions of a source file. They test a constant expression or an identifier to determine which text blocks are passed on to the compiler and which are removed from the source file in the preprocessing stage.

8.4.1 If, Elif, Else, and Endif Directives

Syntax

```
#if restricted-constant-expression
        [ text ]
[ #elif restricted-constant-expression
        text ]
[ #elif restricted-constant-expression
        text ]
            .
            .
            .
            .
[ #else
        text ]
#endif
```

The **#if** directive, together with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. Each **#if** directive in a source file must be matched by a closing **#endif** directive. Zero or more **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif.**

The preprocessor selects one of the given blocks of *text* for further processing. A *text* block is any sequence of text. It can occupy more than one line. Usually the *text* block is program text that has meaning to the compiler or the preprocessor. However, this is not a requirement; the preprocessor can be used to process any kind of text.

The selected *text* is processed by the preprocessor and passed to the compiler. If the *text* contains preprocessor directives, those directives are carried out.

Any text blocks not selected by the preprocessor are removed from the file in the preprocessing stage and are therefore not compiled.

The preprocessor selects a single *text* block by evaluating the *restricted-constant-expressions* following each **#if** or **#elif** directive until a true (nonzero) *restricted-constant-expression* is found. All *text* between the first true *restricted-constant-expression* and the next number sign (#) is selected.

If no *restricted-constant-expression* is true, or if there are no **#elif** directives, the preprocessor selects the text after the **#else** clause. If the **#else** clause is omitted, and no *restricted-constant-expression* in the **#if** block is true, no text is selected.

Each *restricted-constant-expression* follows the rules for restricted-constant-expressions discussed in Section 5.2.10 of Chapter 5, "Expressions and Assignments." Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants, but they can contain the special constant expression "defined(*identifier*)". This constant expression is considered true (nonzero) if the given *identifier* is currently defined. Otherwise, the condition is false (zero). An identifier defined as empty text is considered defined.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. When nested, each **#else**, **#elif**, and **#endif** directive belongs to the closest preceding **#if** directive.

Examples

1.
```
#if defined(CREDIT)
        credit();
#elif defined(DEBIT)
        debit();
#else
        printerror();
#endif
```

2.
```
#if DLEVEL > 5
        #define SIGNAL  1
        #if STACKUSE == 1
                #define STACK   200
        #else
                #define STACK   100
        #endif
#else
        #define SIGNAL  0
        #if STACKUSE == 1
                #define STACK   100
        #else
                #define STACK   50
        #endif
#endif
```

3.
```
#if DLEVEL == 0
        #define STACK 0
#elif DLEVEL == 1
        #define STACK 100
#elif DLEVEL > 5
        display( debugptr );
#else
        #define STACK 200
#endif
```

4.
```
#define REG1    register
#define REG2    register

#if defined(M_86)
        #define REG3
        #define REG4
        #define REG5
#else
        #define REG3    register
        #if defined(M_68000)
                #define REG4    register
                #define REG5    register
        #endif
#endif
```

In the first example, the **#if** and **#endif** directives control compilation of one of three function calls. The function call to *credit* is compiled if the identifier CREDIT is defined. If the identifier DEBIT is defined, the function call to *debit* is compiled. If neither identifier is defined, the call to *printerror* is compiled. Note that CREDIT and *credit* are distinct identifiers in C because their cases are different.

The next two examples assume a previously defined manifest constant, DLEVEL. The second example shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if "DLEVEL $>$ 5" is true. Otherwise, the second set is processed.

In the third example, **#elif** and **#else** directives are used to make one of four choices, based on the value of DLEVEL. The manifest constant STACK is set to 0, 100, or 200, depending on the definition of DLEVEL. If DLEVEL is not defined, "display(debugptr);" is compiled and STACK is not defined.

The fourth example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns **register** storage to variables in the same order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine can accommodate, the compiler honors earlier declarations over later ones. Loss of efficiency can occur if the variables declared later are more heavily used.

The definitions listed above can be used to give priority to the most important register declarations. REG1 and REG2 are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, *b* and *c* have higher priority than *a* or *d*.

```
func(a)

REG3 int a;

{
        REG1 int b;
        REG2 int c;
        REG4 int d;
           .
           .
           .
}
```

When M_86 is defined, the preprocessor removes the REG3 identifier from

the file by replacing it with empty text. This prevents *a* from receiving **register** storage at the expense of *b* and *c*. When M_68000 is defined, all four variables are declared to have **register** storage. When neither M_86 nor M_68000 is defined, *a*, *b*, and *c* are declared with **register** storage.

8.4.2 Ifdef and Ifndef Directives

Syntax

#ifdef *identifier*
#ifndef *identifier*

The **#ifdef** and **#ifndef** directives accomplish the same task as the **#if** directive used with "defined(*identifier*)". These directives can be used anywhere **#if** can be used. These directives are provided only for compatibility with previous versions of the language. The "defined(*identifier*)" constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an **#ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero). Otherwise, the condition is false (zero).

The **#ifndef** directive checks for exactly the opposite condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (zero).

8.5 Line Control

Syntax

#line *constant* ["*filename*"]

The **#line** directive instructs the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename. The compiler uses the internally stored line number and filename to refer to errors encountered during compilation. The line number normally refers to the current input line; the filename refers to the current input file. The line number is increased after each line is processed.

Changing the line number and filename causes the compiler to ignore the previous values and to continue processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of the generated program.

The *constant* value in the **#line** directive is any integer constant. The *filename* can be any combination of characters. It must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous filename remains unchanged.

The current line number and filename are always available through the predefined identifiers __LINE__ and __FILE__. The __LINE__ and __FILE__ identifiers can be used to insert self-descriptive error messages into the program text.

Examples

```
1.  #line 151 "copy.c"

2.  #define ASSERT(cond)          if(!cond)\
    {printf("assertion error line %d, file(%s)\n", \
    __LINE__, __FILE__ );} else ;
```

In the first example, the internally stored line number is set to 151 and the filename is changed to *copy.c*.

In the second example, the macro ASSERT uses the predefined identifiers __LINE__ and __FILE__ to print an error message about the source file if a given "assertion" is not true.

Language Reference

Appendices

Appendices

Language Reference Appendices

A Differences 193

B Syntax Summary 197

Appendix A
Differences

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The differences are listed with cross-references to the corresponding section numbers in *The C Programming Language*.

Section Number in Kernighan and Ritchie	Microsoft C
2.2	Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.
2.3	The identifiers **asm** and **entry** are no longer keywords. New keywords are **const**, **enum** and **void**. (The **const** keyword is not yet implemented but is reserved for future use.) The identifiers **far**, **fortran**, **huge**, **near**, and **pascal** may be keywords, depending on whether the corresponding options are enabled when a program is compiled (see your system documentation).
2.4.1	Hexadecimal and octal constants are treated as unsigned values and are not sign-extended in type conversions.
2.4.3	Hexadecimal bit patterns consisting of a backslash (\), the letter "x", and up to two hexadecimal digits are permitted as character constants (for example, '\x12').

Microsoft C defines two additional escape sequences: the sequence '\v' represents a vertical tab (VT), and the sequence '\"' represents the double quote character.

Character constants always have type **char**, with the result that they are sign-extended in type conversions.

2.6

The **short** type is always 16 bits in length, the **long** type 32 bits. The size of an **int** is machine dependent. On 8086/8088 processors an **int** is 16 bits long, and on 68000 machines it is 32 bits.

4

The **char** type is signed, with the result that a **char** value is sign-extended in type conversions.

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

Microsoft C offers an additional fundamental type, the **enum** (enumeration) type. The **void** type is defined as the return type of functions that do not return a value.

6.5

The keyword **unsigned** can be applied as an adjective to any integer type (**char**, **int**, **short**, or **long**). When **unsigned** stands alone, it is taken to mean **unsigned int**.

6.6

The arithmetic conversions carried out by the Microsoft C Compiler are outlined in Sections 5.3.1 and 5.7 of Chapter 5, "Expressions and Assignments." Although compatible with the Kernighan and Ritchie conversions, the Microsoft C conversions are spelled out in greater detail, including the specific path for each type of conversion.

7.2	In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.
7.14	A structure can be assigned to another structure of the same type.
8.2	The keywords **enum** and **void** are additional type specifiers. Additional acceptable combinations are **unsigned char**, **unsigned short**, **unsigned short int**, **unsigned long**, and **unsigned long int**.
8.4	Optional argument type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.
8.5	Bitfields must be declared **unsigned**.
	The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.
	No relationship exists between the members of two different structure types.
8.6	Unions can be initialized by giving a value for the first member of the union.
9.7	The *expression* of a **switch** statement has **enum** or integral type. Each of the **case** constant-expressions is cast to the type of the *expression*.
12	The number sign (**#**) introducing the preprocessor directive can be preceded by any combination of whitespace characters. Whitespace can also occur between the number sign and the preprocessor keyword.

12.3	The new combination **#if defined**(*identifier*) is intended to supplant the **#ifdef** and **#ifndef** directives. Use of the latter directives is discouraged. The new directive **#elif** (else-if) is designed for use in **#if** and **#if defined** blocks.
14.1	A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions. In expressions involving "->", the expression before the arrow must have the same type (or be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.
17	The listed anachronisms are not recognized.

Appendix B
Syntax Summary

B.1 Tokens 199
B.1.1 Keywords 199
B.1.2 Identifiers 199
B.1.3 Constants 200
B.1.4 Strings 202
B.1.5 Operators 202
B.1.6 Separators 202
B.2 Expressions 203
B.3 Declarations 205
B.4 Statements 208
B.5 Definitions 209
B.6 Preprocessor Directives 210

B.1 Tokens

keyword
identifier
constant
string
operator
separator

B.1.1 Keywords

auto	default	float	register	switch
break	do	for	return	typedef
case	double	goto	short	union
char	else	if	sizeof	unsigned
const†	enum	int	static	void
continue	extern	long	struct	while

The following identifiers may be keywords, depending on whether the corresponding option is enabled when the program is compiled. See your system documentation for details.

far
fortran
huge
near
pascal

B.1.2 Identifiers

identifier:
    letter
    underscore
    identifier letter
    identifier underscore
    identifier digit

---

† Not yet implemented.

letter: one of
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

underscore:
_

digit: one of
0 1 2 3 4 5 6 7 8 9

B.1.3 Constants

constant:
integer-constant
long-constant
floating-point-constant
char-constant
enum-constant

integer-constant:
0
decimal-constant
octal-constant
hexadecimal-constant

decimal-constant:
nonzero-digit
decimal-constant digit

nonzero-digit: one of
1 2 3 4 5 6 7 8 9

octal-constant:
0*octal-digit*
octal-constant octal-digit

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-constant:
0x*hexadecimal-digit*
0X*hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

long-constant:
integer-constant l
integer-constant L

floating-point-constant:
fractional-constant exponent
fractional-constant
digit-seq exponent

fractional-constant:
digit-seq . digit-seq
. digit-seq
digit-seq .

digit-seq:
digit
digit-seq digit

exponent:
e *sign digit-seq*
E *sign digit-seq*
e *digit-seq*
E *digit-seq*

sign:
+
−

char-constant:
'char'

char:
rep-char
escape-sequence

rep-char:
Any single representable character except the single quote ('), backslash (\), or newline character.

escape-sequence: one of
\' \" \\ *ddd* \x*ddd* \b
\f \n \r \t \v

enum-constant:
    identifier

B.1.4 Strings

string-literal:
    " "
    " *char-seq*"

char-seq:
    char
    char-seq char

B.1.5 Operators

operator: one of

!	~	++	--	+
-	*	/	%	<<
>>	<	<=	>	>=
==	!=	|	&	^
&&	||	=	+=	-=
*=	/=	%=	>>=	<<=
&=	^=	|=	?:	,
[]	()	.	->	

B.1.6 Separators

separator: one of

[	]	(	)	{	}
*	,	:	=	;	#

B.2 Expressions

In this section (B.2) the brackets shown are part of the syntax for the language and should be interpreted literally.

expression:
    identifier
    constant
    string
    expression(*expression-list*)
    expression()
    expression[*expression*]
    expression.*identifier*
    expression−>*identifier*
    unary-expression
    binary-expression
    ternary-expression
    assignment-expression
    (*expression*)
    (*type-name*)*expression*
    constant-expression

expression-list:
    expression
    expression-list, *expression*

unary-expression:
    unop expression
    sizeof(*expression*)

unop: one of
    − ~ ! * &

lvalue:
    identifier
    expression[*expression*]
    expression.*expression*
    expression−>*expression*
    **expression*
    (*type-name*)*expression*
    (*lvalue*)

type-name:
See Section B.3, "Declarations."

binary-expression:
expression binop expression

binop: one of

*	/	%	+	-
<<	>>	<	>	<=
>=	==	!=	&	|
^	&&	||	,	

ternary-expression:
expression ? *expression* : *expression*

assignment-expression:
lvalue++
lvalue--
++*lvalue*
--*lvalue*
lvalue assignment-op expression

assignment-op: one of

=	*	/=	%=	+=	-=
<<=	>>=	&=	|=	^=	

constant-expression:
identifier
constant
(*type-name*)*constant-expression*
unary-expression
binary-expression
ternary-expression
(*constant-expression*)

B.3 Declarations

In this section (B.3) the brackets shown are part of the syntax for the language and should be interpreted literally.

declaration:
    sc-specifier type-specifier declarator-list;
    type-specifier declarator-list;
    sc-specifier declarator-list;
    type-specifier;
    typedef *type-specifier declarator-list*;

sc-specifier:
    auto
    extern
    register
    static

type-specifier:
    char
    double
    enum-specifier
    float
    int
    long
    long int
    short
    short int
    struct-specifier
    typedef-name
    union-specifier
    unsigned
    unsigned char
    unsigned int
    unsigned long
    unsigned long int
    unsigned short
    unsigned short int

enum-specifier:
    enum *tag* {*enum-list*}
    enum {*enum-list*}
    enum *tag*

tag:
    identifier

enum-list:
    enumerator
    enum-list , *enumerator*

enumerator:
    identifier
    identifier = *constant-expression*

struct-specifier:
    struct *tag* {*member-declaration-list*}
    struct {*member-declaration-list*}
    struct *tag*

member-declaration-list:
    member-declaration
    member-declaration-list member-declaration

member-declaration:
    type-specifier declarator-list;
    type-specifier identifier : *constant-expression*;
    type-specifier : *constant-expression*;

declarator-list:
    declarator
    declarator = *initializer*
    declarator-list , *declarator*

declarator:
    identifier
    declarator[ ]
    declarator[*constant-expression*]
    **declarator*
    declarator( )
    declarator(*arg-type-list*)
    (*declarator*)

arg-type-list:
    type-name
    arg-type-list, *type-name*
    arg-type-list,
    void

    ,
    void*

type-name:
    type-specifier
    type-specifier abstract-declarator

abstract-declarator:
    *
    [ ]
    (*arg-type-list*)
    **abstract-declarator*
    *abstract-declarator**
    abstract-declarator[ ]
    abstract-declarator[*constant-expression*]
    [ ] *abstract-declarator*
    [*constant-expression*] *abstract-declarator*
    abstract-declarator( )
    abstract-declarator(*arg-type-list*)
    (*abstract-declarator*)

initializer:
    expression
    {*initializer-list*}

initializer-list:
    initializer
    initializer-list, *initializer*

typedef-name:
    identifier

union-specifier:
    union *tag* {*member-declaration-list*}
    union {*member-declaration-list*}
    union *tag*

B.4 Statements

In this section (B.4) brackets enclose optional portions of the syntax.

statement:
    break;
    case *constant-expression* : *statement*
    compound-statement
    continue;
    default : *statement*
    do *statement* **while** (*expression*);
    expression;
    for ([*expression*];[*expression*];[*expression*])*statement*;
    goto *identifier*;
    identifier : *statement*
    if (*expression*) *statement* [**else** *statement*]
    ;
    return [*expression*];
    switch (*expression*) *statement*
    while (*expression*) *statement*

compound-statement:
    {[*declaration-list*] [*statement-list*]}

declaration-list:
    declaration
    declaration-list declaration

statement-list:
    statement
    statement-list statement

B.5 Definitions

In this section (B.5) brackets enclose optional portions of the syntax.

definition:
    function-definition
    data-definition

function-definition:
    [*sc-specifier*] [*type-specifier*] *declarator* ([*parameter-list*]) [*parameter-decs*] *compound-statement*

parameter-list:
    identifier
    parameter-list , *identifier*

parameter-decs:
    declaration
    declaration-list declaration

data-definition:
    declaration

B.6 Preprocessor Directives

In this section (B.6) brackets enclose optional portions of the syntax.

directive:
    #
    #**define** *identifier*[([*parameter-list*])] [*token-seq*]
    #**elif** *restricted-constant-expression*
    #**else**
    #**endif**
    #**if** *restricted-constant-expression*
    #**ifdef** *identifier*
    #**ifndef** *identifier*
    #**include** <*string*>
    #**include** "*string*"
    #**line** *digit-seq*
    #**line** *digit-seq string*
    #**undef** *identifier*

token-seq:
    token
    token-seq token

restricted-constant-expression:
    defined (*identifier*)
    Any *constant-expression* except for **sizeof** expressions, casts, and enumeration constants.

Index

Language Reference Index

! (logical NOT) operator, 99
!= (inequality) operator, 108
(number sign), 177
% (remainder) operator, 102
& (address-of) operator, 100
& (bitwise AND) operator, 110
&& (logical AND) operator, 112
( ) (parentheses)
  in expressions, 95
  in function calls, 89
* (asterisk)
  indirection operator, 100
  multiplication operator, 102
  pointer modifier, 48, 64
+ (addition) operator, 104
++ (increment) operator, 117
, (comma)
  in arg-type-list, 66
  sequential evaluation operator, 114
- (arithmetic negation) operator, 99
- (subtraction) operator, 104
-- (decrement) operator, 117
  -> (member selection) operator, 92, 196
. (member selection) operator, 92
/ (division) operator, 102
< (relational) operator, 108
<< (left shift) operator, 107
<= (relational) operator, 108
= (simple assignment) operator, 118
== (equality) operator, 108
> (relational) operator, 108
>= (relational) operator, 108
>> (right shift) operator, 107
? : (conditional) operator, 115
[ ] (brackets)
  as array modifier, 48
  in array declarations, 62
  in subscript expressions, 90
  notational conventions, 6
\ (backslash) character, 14, 15
^ (bitwise exclusive OR) operator, 110
{ } (braces)
  in initialization, 77
| (bitwise inclusive OR) operator, 110
| | (logical OR) operator, 112
~ (bitwise complement) operator, 99

Abstract declarator, 83
Actual arguments, 170
  conversion, 170
  order of evaluation, 167
  passing, 170
  pointer, 170
  side effects, 167
  type-checking, 170
  variable number, 172
Addition operator (+), 104
Additive operators, 104
Address-of (&) operator, 100
Aggregate types, 43
  array, 62
  initialization, 76, 77
  structure, 57
  union, 60
Anachronisms, 196
AND operator
  bitwise (&), 110
  logical (&&), 112
Angle brackets (<>), 183
argc parameter, 33
Argument type list, 66, 67, 165, 195
  void keyword, 67
  void *, 67
Argument type-checking, 170
Arguments
  actual. *See* Actual arguments.
  command line, 32
  conversion, 170
  formal, 161
  order of evaluation, 167
  passing, 170
  pointer, 170

Arguments *(continued)*
  side effects, 167
  to main function, 32
  type-checking, 170
  variable number, 66, 172
argv parameter, 33
Arithmetic conversions, 97, 194
Arithmetic negation operator (−), 99
Arithmetic with pointers, 105
Array
  declarations, 62
  elements, 90
  identifiers, 88
  initialization, 79
  modifier, 48, 62
  multidimensional, 62
  references, 90, 91
  storage, 63
  types, 48, 62
asm, 192
Assignment conversions, 124
Assignment expressions, 94
Assignment operators, 116
  compound, 118
  simple (=), 118
Associativity
  modifiers, 49
  operators, 119
Asterisk (*)
  indirection operator, 100
  multiplication operator, 102
  pointer modifier, 48, 64
auto storage class specifier, 69
  with internal variables, 73
auto variables
  initialization, 76

Backslash character (\), 14, 15
Binary expressions, 94
Binary operators, 97
Bitfields, 58, 195
Bitwise AND operator (&), 110
Bitwise complement operator (~), 99
Bitwise exclusive OR operator (^), 110
Bitwise inclusive OR operator (|), 110
Block
  defined, 33
Body
  function, 164
Braces ({})
  in initialization, 77
Brackets ([])
  as array modifier, 48
  in array declarations, 62
  in subscript expressions, 90
  notational conventions, 6
Branch statements
  if, 145
  switch, 150, 195
break statement, 135
Byte
  size of, 195

C character set, 11
Calls. *See* Function calls.
case constant expression, 150
Case
  significance of, 12, 23, 24
Casts. *See* Type-casts.
char type, 44, 194
  range of values, 46
  storage, 46
Character constants, 20, 194
Character set
  C, 11
  representable, 11
Character types. *See* Integral types.
Characters
  backslash (\), 14, 15
  continuation (\), 15
  escape sequence, 14, 193
  number sign, 177
  punctuation, 13
  special, 13
  whitespace, 12
Comma (,)
  in arg-type-list, 66
Comma operator. *See* Sequential evaluation operator.
Command line arguments, 32
Comments, 25
Complement operators, 99
Complex declarators, 49
  interpreting, 49

Compound assignment operators, 118
Compound statement, 136
Conditional compilation, 184, 188
Conditional operator (? :), 115
Conditional statements
  if, 145
  switch, 150, 195
const, 193
Constant-expressions, 95
  defined (identifier), 185
  case, 150
  in preprocessor directives, 95, 185
  in switch statement, 150, 195
  initializers, 95
  restricted, 95, 185
Constants, 4, 88
  character, 20, 193, 194
  floating-point, 19
  integer, 17
    decimal, 17
    hexadecimal, 17, 193
    long, 18
    negative, 18
    octal, 17, 193
  manifest (symbolic), 177
  string literals, 21
Continuation character (\), 15
continue statement, 138
Conventions
  notational, 5
Conversions
  actual arguments, 170
  arithmetic, 97, 194
  assignment, 124
  formal parameters, 171
  from enumeration types, 129
  from floating-point types, 128
  from pointer types, 129
  from signed integral types, 124
  from unsigned integral types, 126
  function-call, 130, 170
  operator, 130
  type-cast, 129
  usual arithmetic, 97, 194
CTRL-Z character, 12

Data types. *See* Types.

Declarations, 29
  array, 62
  external, 69, 70, 75
  form of, 43
  formal parameters, 161
  forward, 66, 165
  function, 29, 66, 157, 165, 195
    storage class, 75
    with variable number of arguments, 66
  internal, 69, 73
  pointer, 64
  typedef, 80, 81
  variable, 29, 53
    array, 62
    enum, 55
    multidimensional arrays, 62
    pointer, 64
    simple, 54
    structure, 57
    union, 60
Declarators, 48
  abstract, 83
  complex, 49
  in parentheses, 49
Decrement operator (--), 117
default keyword, 150
Default storage class
  external variable declarations, 70
  function declarations, 75, 165
  internal variable declarations, 73
#define directive, 178
defined (identifier). *See* #if directive.
Definitions, 29
  function, 29, 157
  variable, 29, 70, 73
Differences, 193
Directives, 29, 177, 195
  #define, 178
  #elif, 184, 194
  #else, 184
  #endif, 184
  #if, 184, 196
  #ifdef, 188, 196
  #ifndef, 188, 196
  #include, 182
  lifetime, 33
  #line, 188

Directives *(continued)*
restricted constant-expressions, 95
#undef, 181
Division operator (/), 102
do statement, 139
continuing execution, 138
terminating execution, 135
double type, 44
range of values, 46
storage, 46

Elements, 90
#elif directive, 184, 185, 196
Ellipses
notational conventions, 6
#else directive, 184
End-of-file indicator, 12
#endif directive, 184
entry, 193
enum, 193, 195
enum types, 44, 55, 80, 195
range of values, 46
storage, 46, 55
enum-list, 55
Enumeration
constants
naming class, 38
declarations, 55, 80
identifiers, 88
set, 55
tags, 55, 80
naming class, 38
types, 44
envp, 33
Equality operator (==), 108
Escape sequences, 14, 193, 194
Evaluation order. *See* Order of evaluation.
Examples
notational conventions, 8
Execution
starting point, 32
Exit
from functions, 148
from switch statement, 150
Expression list, 89
Expression statements, 140
Expressions, 87
assignment, 94
binary, 94
constant. *See* Constant-expressions.
enum, 88
floating-point, 88
function call, 90
in parentheses, 95
integral, 88
lvalue, 116
member selection, 92
pointer, 88
struct, 88
subscript, 90
ternary, 94
type-cast, 95
unary, 94
union, 88
with operators, 94
extern storage class specifier, 69
with external variables, 70
with function declarations, 75, 158, 165
with internal variables, 73
External level declarations, 69, 70, 75

far keyword, 64, 193
Filename
changing, 188
float type, 44
range of values, 46
storage, 46
Floating-point
constants, 19
expressions, 88
identifiers, 88
types, 44
converting, 128
for statement, 141
continuing execution, 138
terminating execution, 135
Formal parameters, 161
conversion, 171
declaring, 161
identifiers, 161
naming class, 37
storage class, 161

Formal parameters *(continued)*
type-checking, 161, 171
fortran, 193
Forward declarations, 165
Function body, 164
Function calls, 89, 157, 167
conversions, 130, 170
expression, 90
indirect, 167
recursive, 174
type-checking, 170
with variable number of arguments, 172
Function declarations, 29, 66, 157, 165, 195
extern, 165
forward, 165
implicit, 165
static, 165
storage class specifier, 75, 165
with variable number of arguments, 66, 172
Function definitions, 29, 157
extern, 158
return type, 158
static, 158
storage class specifier, 158
Function identifiers, 89
Function modifier, 48
Function pointers, 167
Function returning type, 48, 66
Function-call conversions, 130, 170
Functions, 157
calling, 167
exit from, 148
extern, 158, 165
main, 32
naming class, 37
parameters, 32
return type, 158
implicit, 165
return value, 148, 164
static, 158, 165
storage class, 158, 165
visibility, 158, 165
Fundamental types, 44
char, 44
double, 44
Fundamental types *(continued)*
enum, 44, 192
float, 44
floating-point, 44
initialization, 77
int, 44
integral, 44
long, 44
range of values, 46
short, 44
storage, 46
unsigned char, 44
unsigned int, 44
unsigned long, 44
unsigned short, 44

Global lifetime, 33, 69
Global variables, 34
initialization, 76
references to, 74
Global visibility, 33
goto statement, 143

Hexadecimal escape sequences, 14, 193
Hexadecimal integer constants, 17
huge, 193

Identifiers, 23, 88, 193
array, 88
enum, 88
floating-point, 88
formal parameters, 161
function, 89
integral, 88
modified, 48
naming classes, 37
pointer, 88
predefined, 189
struct, 88
union, 88
#if defined(identifier). *See* #if directive.
#if directive, 184, 185, 195
if statement, 145
#ifdef directive, 188, 194
#ifndef directive, 188, 194

Implicit function declarations, 165
Implicit return type, 165
#include directive, 182
Include files, 182
Increment operator (++), 117
Indirection operator (*), 100
Inequality operator (!=), 108
Initialization, 76
  aggregate types, 76, 77
  array variables, 79
  auto variables, 76
  fundamental types, 77
  global variables, 76
  initial values, 76
  register variables, 76
  restrictions, 76
  static variables, 76
  with constant-expressions, 95
  with string literals, 79
int type, 44, 194
  range of values, 46
  storage, 46
Integer constants
  decimal, 17
  hexadecimal, 17, 193
  long, 18
  negative, 18
  octal, 17, 193
  unsigned, 193
Integer types. *See* Integral types.
Integral expressions, 88
Integral identifiers, 88
Integral types, 44
  converting, 124, 126
Internal level variable declarations, 69, 73
Italics
  notational conventions, 5
Iterative statements
  do, 139
  for, 141
  while, 153

Keywords, 8, 24, 193, 195

Labeled statements, 143
Labels, 143
  case, 150
  default, 150
  naming class, 38
Left shift operator (<<), 107
Lifetime
  defined, 33
  global, 33, 69
  local, 33, 69
Line control, 188
#line directive, 188
Line numbers
  changing, 188
Linked lists, 58
Local lifetime, 33, 69
Local variables, 34
Logical AND operator (&&), 112
Logical NOT operator (!), 99
Logical operators, 112
  order of evaluation, 112
Logical OR operator (||), 112
long type, 44, 194
  range of values, 46
  storage, 46
Loops
  do statement, 139
  for statement, 141
  while statement, 153
Lvalue expressions, 116

Macros, 177
Main function, 32
  parameters, 32
Manifest constants, 177
  removing definitions, 181
member-declaration-list, 57, 80
Member selection operators (-> and .), 92, 196
Members
  bitfields, 58
  naming class, 38
  referring to, 92
  structure, 57
  union, 60
Modifiers, 48
  array, 48, 62
  associativity, 49

Modifiers *(continued)*
function, 49
pointer, 49, 64
precedence, 49
Multidimensional arrays, 62
references to, 91
Multiplication operator (*), 102

Names. *See* Identifiers *or* Type names.
Naming classes, 37
structures, 195
unions, 195
near keyword, 64, 193
Nested visibility, 34
Notational conventions, 5
Null statement, 147
Number sign (#), 177

Octal escape sequences, 14
Octal integer constants, 17
Operands, 87
Operator conversions, 130
Operators, 94, 97
addition (+), 104
address-of (&), 100
arithmetic negation (−), 99
assignment, 116
associativity of, 119
binary, 97
bitwise AND (&), 110
bitwise complement (~), 99
bitwise exclusive OR (^), 110
bitwise inclusive OR (|), 110
complement, 99
compound assignment, 118
conditional (? :), 115
decrement (−), 117
division (/), 102
equality (==), 108
increment (++), 117
indirection (*), 100
inequality (!=), 108
left shift (<<), 107
list of, 16
logical, 112
logical AND (&&), 112
logical not (!), 99
logical OR (||), 112
member selection (−> and .), 92, 196
multiplication (*), 102
precedence, 119
relational (<, <=, >, >=), 108
remainder (%), 102
right shift (>>), 107
sequential evaluation (,), 114
shift, 107
simple assignment (=), 118
sizeof, 101
subtraction (−), 104
ternary (? :), 97, 115
unary, 97
OR operator
bitwise exclusive (^), 110
bitwise inclusive (|), 110
logical (||), 112
Order of evaluation, 87, 121
actual arguments, 167
logical operators, 112
Overview, 3

Parameters
argc, 33
argv, 33
conversion, 171
envp, 33
formal, 161
conversion, 171
declaring, 161
identifiers, 161
naming class, 37
storage class, 162
type-checking, 161
main function, 32
type-checking, 171
Parentheses
as function modifier, 48
in complex declarators, 49
in expressions, 95
in function calls, 89
in macros, 181
pascal, 193
Passing by reference, 167, 170

Passing by value, 170
Period (.) in member selection
expressions, 92
Pointer
arguments, 167
arithmetic, 105
declarations, 64
expressions, 88
identifiers, 88
modifier, 48, 64
to function, 167
types, 48, 64
converting, 129
storage, 64
Pound sign (#). *See* Number sign.
Precedence
modifiers, 49
operators, 119
Predefined identifiers, 189
Preprocessor directives. *See* Directives.
Program execution, 32
Program structure, 29
Programming examples
notational conventions, 8
Punctuation characters, 13

Quotation marks, 7

Range of values, 46
Recursion, 174
References
to arrays, 90, 91
to global variables, 74
register storage class specifier, 69
with internal variables, 73
register variables
initialization, 76
Relational operators (<, <=, >, >=), 108
Remainder operator (%), 102
Removing macro definitions, 181
Removing manifest constant definitions, 181
Representable character set, 11
Representation
internal, 47
Reserved words. *See* Keywords.
Restricted-constant-expression, 95, 185
return statement, 148
Return type, 67
declaring, 165
implicit, 165
in function definitions, 158
Return value, 148, 164, 196
Returning control, 148
Right shift operator (>>), 107

Selection statements
if, 145
switch, 150
Separators, 202
Sequential evaluation operator (,), 114
Shift operators, 107
short type, 44, 194
range of values, 46
storage, 46
Side effects, 87, 123
in actual arguments, 167
in macros, 181
Signed integer types
converting, 124
Simple assignment operator (=), 118
Simple variable declarations, 54
sizeof operator, 101
Source files, 29, 30
Special characters, 13
Statement body, 133
compound, 136
Statement labels, 143
naming class, 38
Statements, 133
break, 135
compound, 136
continue, 138
do, 139
expression, 140
for, 141
goto, 143
if, 145
labeled, 143
null, 147
return, 148
switch, 150, 193

Statements *(continued)*
while, 153
static storage class specifier, 69
in forward declarations, 165
in function declarations, 75, 165
in function definitions, 158
with external level variables, 70
with internal level variables, 75
static variables
initialization, 76
Storage
arrays, 63, 92
char type, 46
double type, 46
enum type, 46, 55
float type, 46
global, 69
int, 46
local, 69
long type, 46
pointers, 64
short type, 46
structure variables, 58
union variables, 60
unsigned char type, 46
unsigned int type, 46
unsigned long type, 46
unsigned short type, 46
void type, 46
Storage class specifiers, 69
auto, 69
with internal variables, 73
default, 70, 73, 75, 165
extern, 69
with external variables, 70
with function declarations, 75
with internal variables, 73
formal parameters, 162
in forward declarations, 165
in function declarations, 165
in function definitions, 158
register, 69
with internal variables, 73
static, 69
with external variables, 70
with function declarations, 75
with internal variables, 73
Storage classes, 69
String initializers, 79
String literals, 21, 79, 89
Strings. *See* String literals.
struct type specifier, 57, 80
Structure
assignment, 195, 196
declarations, 57
expressions, 88
identifiers, 88
members, 57
bitfield, 58
naming class, 38
referring to, 92
passing, 196
returning, 196
tags, 57, 80
naming class, 38
types, 57, 80
storage, 58
Subscript expressions, 90
Subtraction operator (−), 104
switch statement, 150, 195
exit from, 150
terminating execution, 135
Symbolic constants. *See* Manifest constants.
Syntax conventions. *See* Notational conventions.
Syntax summary, 199

Tags
enumeration, 55, 80
naming class, 38
structure, 57, 80
union, 80
Ternary expressions, 94
Ternary operator (? :), 97, 115
Tokens, 26, 199
Transfer statements
break, 135
continue, 138
goto, 143
labeled statements, 143
Two's complement operator, 99
Type conversions. *See* Conversions.
Type declarations, 80
Type list argument. *See* Argument type list

Type names, 82
  in arg-type-list, 66
  in function declarations, 66
  void, 171
Type specifiers, 44
  abbreviations, 45
  char, 44, 80, 194
  double, 44
  enum, 44, 55, 194
  float, 44
  int, 44, 194
  long, 44, 194
  short, 44, 194
  struct, 57, 80
  union, 60, 80
  unsigned char, 44, 194, 195
  unsigned int, 44
  unsigned long, 44, 194, 195
  unsigned long int, 194, 195
  unsigned short, 44, 194, 195
  unsigned short int, 44, 194, 195
  void, 44, 194
Type casts, 95, 129
Type-checking, 171
  actual arguments, 67, 170
  formal parameters, 171
typedef
  declarations, 81
  naming class, 38
  types, 81
Types
  aggregate, 43
  array, 48, 62
  char, 44, 194
  defining, 80
  double, 44
  enum, 44, 55, 80, 194
  float, 44
  floating-point, 44
  function returning, 48, 66
  fundamental, 44
  int, 44, 194
  integral, 44
  long, 44, 194
  names of, 82
  pointer, 48, 64
  short, 44, 194
  struct, 57, 80
Types *(continued)*
  typedef, 81
  union, 60, 80
  unsigned, 44, 194, 195
  user-defined, 80
  void, 194

Unary expressions, 94
Unary operators, 97
#undef directive, 181
Union
  declarations, 60, 80
  expressions, 88
  identifiers, 88
  members
    naming class, 38
    referring to, 92
  tags, 80
    naming class, 38
  types, 60, 80
unsigned types, 44, 194, 195
  converting, 126
  range of values, 46
  storage, 46
User-defined types, 80
Usual arithmetic conversions, 97, 194

Variable declarations, 29, 53
  array, 62
  enum, 55
  external, 69, 70
  internal, 69, 73
  pointer, 64
  simple, 54
  structure, 57
  union, 60
Variable definitions, 29, 70
Variable names. *See* Identifiers.
Variables
  array, 62, 79
  auto, 76
  enumeration, 55
  global, 34
  local, 34
  naming class, 37
  pointer, 64

Variables *(continued)*
register, 76
static, 76
structure, 57
union, 60
Visibility
defined, 33
function, 158, 165
global, 33
nested, 34
void, 67
in argument type list, 67
type name, 171
type specifier, 44, 194
void * construction, 67

while statement, 153
continuing execution, 138
terminating execution, 135
Whitespace characters, 12

Manual Report Form

Please tell Microsoft what you think about these manuals. Your comments and suggestions help us improve our products. Mail this questionnaire to:

MICROSOFT Corporation
10700 Northup Way, Building #4
Box 97200
Bellevue, WA 98009
Attn: User Education, Systems Languages

Respond to any or all of the following questions. If you want to make additional comments, use the back of this page or a separate page.

Product Name: **Version Number:**

1. Did you find errors in the manuals? Please give the page number and description.

2. Which sections of the manuals do you consider most important? Do you have any suggestions for improving these sections?

3. Is it easy to find the information you need? Is anything missing?

4. Are the manuals clear and easy to read? Please name specific sections that you found confusing.

5. What type of user are you?
   _____ First-time programmer
   _____ First-time programmer in this language, but experienced in at least one other language
   _____ Experienced programmer

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

MICROSOFT®